ACPI Implementers' Guide

Intel

Microsoft

Toshiba

February 1, 1998

Draft

INTELLECTUAL PROPERTY DISCLAIMER

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION, OR SAMPLE.

NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED OR INTENDED HEREBY.

INTEL, MICROSOFT, AND TOSHIBA, DISCLAIM ALL LIABILITY, INCLUDING LIABILITY FOR INFRINGEMENT OF PROPRIETARY RIGHTS, RELATING TO IMPLEMENTATION OF INFORMATION IN THIS SPECIFICATION. INTEL, MICROSOFT, AND TOSHIBA, DO NOT WARRANT OR REPRESENT THAT SUCH IMPLEMENTATION(S) WILL NOT INFRINGE SUCH RIGHTS.

THIS DOCUMENT IS A DRAFT FOR COMMENT ONLY AND IS SUBJECT TO CHANGE WITHOUT NOTICE. READERS SHOULD NOT DESIGN PRODUCTS BASED ON THIS DOCUMENT.

Revisions

The Draft Revision 0.4 of the *ACPI Implementers' Guide* (dated March 19, 1997) was the first release of the *Guide* and is the baseline for all revisions documented in the following table.

Date	Revision number	Change	Contributor/Editor
4/2/97	0.41	In section 3.4.1, "Main File of Desktop Concept Machine Sample Code," in the PX43 Device object code, deleted the OperationRegion named CFG3 and added an instructive comment to the Operation Region named GPOB. In section 3.4.2, "SuperIO ASL Include File," corrected the Offset term values. This same correction also had to be made to section 3.3.4, "Operation Region and Field Definitions for a Super I/O Chip."	Randall_Scott@ccm.fm.intel.com
4/2/97	0.41	Corrected ASL code examples to correctly use ASL specification naming convention that first argument in argument list received by a called control method is named Arg0, second argument in list (if any) is named Arg1, and so on.	markwi@microsoft.com
6/14/97	0.42	Made corrections to various sections of the Guide, based on e-mail from ACPI implementers.	markwi@microsoft.com
6/14/97	0.43	Added the new section 5.1.3, Trajan Thermal Control.	Wilson_Huang@ccm.sc.intel.com
6/15/97	0.44	In section 5.1.2, Trajan Interrupt Structure, replaced all the ASL sample code for PCI interrupt routing with a new version.	Phil_Mummah@ccm.sc.intel.com
6/16/97	0.45	In section 5.3.1, Device Power Management, added example ACPI name spaces that show Device objects for IDE controllers and drives on both the PCI bus and the ISA bus.	Phil_Mummah@ccm.sc.intel.com
6/17/97	0.46	In section 2.6, Complete Listing of the Mobile Concept Machine ASL Code, changed the code inside the BAT0 Device object.	tommy_yamasaki@smtpgate.tais.com
6/17/97	0.47	Deleted the Appendices.	markwi@microsoft.com

Date	Revision number	Change	Contributor/Editor
6/17/97	0.48	In section 5.1.2, Trajan Interrupt Structure, redrew the diagram.	markwi@microsoft.com
10/1/97	0.49	Added first draft of chapter about implementing Name Space.	tommy_yamasaki@smtpgate.tais.com/ a-gsoler@microsoft.com
10/1/97	0.49	Added first draft of chapter about implementing GPEs.	a-gsoler@microsoft.com
10/1/97	0.49	Added first draft of chapter about implementing ECs on mobile platforms.	Wilson_Huang@ccm.sc.intel.com/ markwi@microsoft.com
10/1/97	0.49	Added first draft of chapter about AMLI debugger.	markwi@microsoft.com
10/1/97	0.49	Substituted location (on CD and URL to Web) for latest sample ASL code for Mobile, Desktop, and Server concept machines; removed the ASL code listings from the ImpGuide.	markwi@microsoft.com
10/1/97	0.49	Made changes to the example ASL code in the BIOS case study chapter (Trajan).	Wilson_Huang@ccm.sc.intel.com
10/1/97	0.49	Moved the Glossary of Terms from the Introduction chapter to a new chapter at the end of the ImpGuide.	markwi@microsoft.com
2/1/98	0.50	Updated ASL code in for Mobile, Server, Desktop, and Trajan Chapters. Available on the ACPI website at http://www.teleport.com/~acpi	markwi@microsoft.com
2/1/98	0.50	Updated Mobile Concept Machine Chapter and NameSpace ImpGuide to reflect most recent *Errata* items.	tommy_yamasaki@smtpgate.tais.com/ a-gsoler@microsoft.com
2/1/98	0.50	Corrected Server Concept Machine ASL code to reflect IndexField terms for Operation Region and Field Declarations for a Super I/O Chip working registers (see 10.3.3.)	Mikets@microsoft.com
2/1/98	0.50	Added first draft of chapter on docking.	Robmck@microsoft.com

Contents

1. Introduction

Note: This is the February 1, 1998, version of the *ACPI Implementers' Guide*; the latest version of the *Guide* is always available at www.teleport.com/~acpi.

The *ACPI Implementers' Guide* is a practical guide for engineers that are working to get an ACPI-compatible system design up and running. This *Guide* is designed to work with the *ACPI Specification, Version 1.0*, and the *ACPI Specification, Version 1.0, Errata* publications; you need those other two publications nearby to effectively use this *Guide*.

1.1 Structure of the ACPI Implementers' Guide

This *Guide* has the following sections:

Section Title	Description	Suggested Use
ACPI Name Space ImpGuide	Describes how various name spaces are dealt with in an ACPI system and primes the reader on basic concepts involved in the Mobile Concept machine.	If you are designing and building a mobile platform, use this chapter to further your understanding of basic concepts and requirements as well as learn the requirements for name spaces in an ACPI system.
ACPI Event ImpGuide	Describes how to implement events in ACPI, including power management event signals, and runtime event signals using the PIIX4 chipsets.	Use this chapter to learn about various methods for implementing runtime and wake events in ACPI.
ACPI Embedded Controller ImpGuide	Describes the advantages of using an Embedded Controller (EC) on an ACPI platform and gives several examples of EC use on a mobile platform (based on the use of an EC on the Trajan platform).	If you are planning to use an EC on your ACPI platform, or are implementing an EC, use this chapter for design and implementation ideas.
Installing and Using the AMLI Debugger	Describes how to install and use the AMLI debugger under both Windows 98 and Windows NT.	If you are debugging AML code and want to use the AMLI debugger, refer to this chapter.
ACPI Mobile Concept Machine	Describes a hypothetical mobile platform design with a hardware block diagram, models the hardware with objects in ACPI name space, and fills the objects with ASL code. Focuses on • Using an embedded controller. • Managing removable devices coming and going in a bay. • Managing docking and undocking events.	If you are designing and building a mobile platform that has an ACPI-compatible chipset, use this chapter to get ideas for your design and to get blocks of sample ASL code you can modify instead of writing all your ASL code from scratch.
ACPI BIOS Case Study	This section goes one step beyond the concept machine sections. This section uses the ACPI-compatible	Use this chapter both to double-check the validity of your ASL code and to package your ACPI

Section Title	Description	Suggested Use
	Trajan 430TX motherboard as a case study and shows how to build an ACPI BIOS for an actual ACPI-compatible motherboard product.	BIOS.
ACPI Desktop Concept Machine	Describes a hypothetical desktop platform design with a hardware block diagram, models the hardware with objects in ACPI name space, and fills the objects with ASL code. Focuses on • PowerResource implementation. • Thermal Zone implementation. • Power Button implementation.	If you are designing and building a Desktop that has an ACPI-compatible chipset, use this chapter to get ideas for your design and to get blocks of sample ASL code you can modify instead of writing all your ASL code from scratch.
ACPI Server Concept Machine	Describes a hypothetical server platform design with a hardware block diagram, models the hardware with objects in ACPI name space, and fills the objects with ASL code. Focuses on • Managing removable disk drives.	If you are designing and building a Server that has an ACPI-compatible chipset, use this chapter to get ideas for your design and to get blocks of sample ASL code you can modify instead of writing all your ASL code from scratch.
Using the ACPI Embedded Controller and SMBus Interfaces	Presents sample ACPI Embedded Controller Interface and ACPI SMBus interface implementations and walks through the transaction protocol across these buses for each of these implementations.	Use to better understand these two ACPI interfaces.
Definition of Terms	Provides definitions of general ACPI terms	Refer to this chapter whenever you encounter a term you are unfamiliar with or need to learn more about.

2. ACPI Name Space ImpGuide

This section describes how various name spaces are handled in an ACPI system. Figure 1 is a block diagram of the concept mobile platform ACPI system.

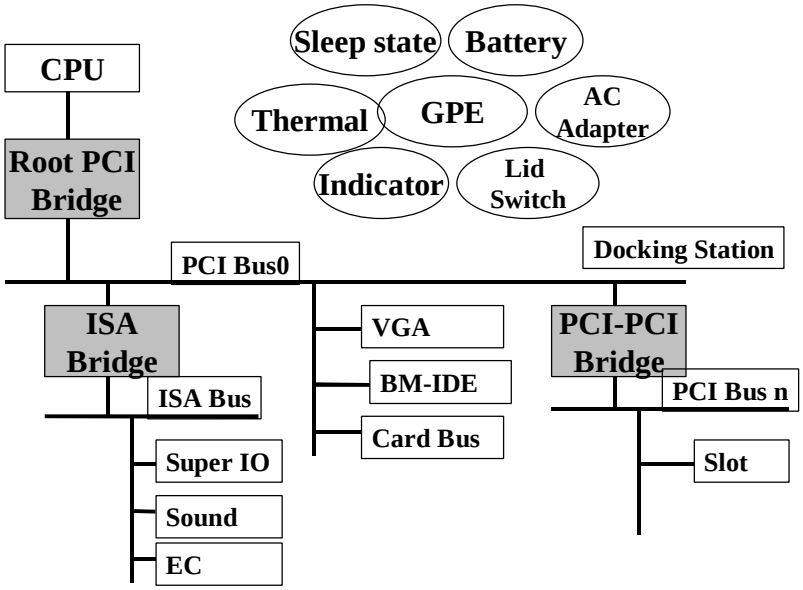


Figure 1. ACPI Block Diagram for Mobile Concept Machine

2.1 Basic Name Space Structure

The name space uses the following structure in ASL, as illuminated by Figure 2.

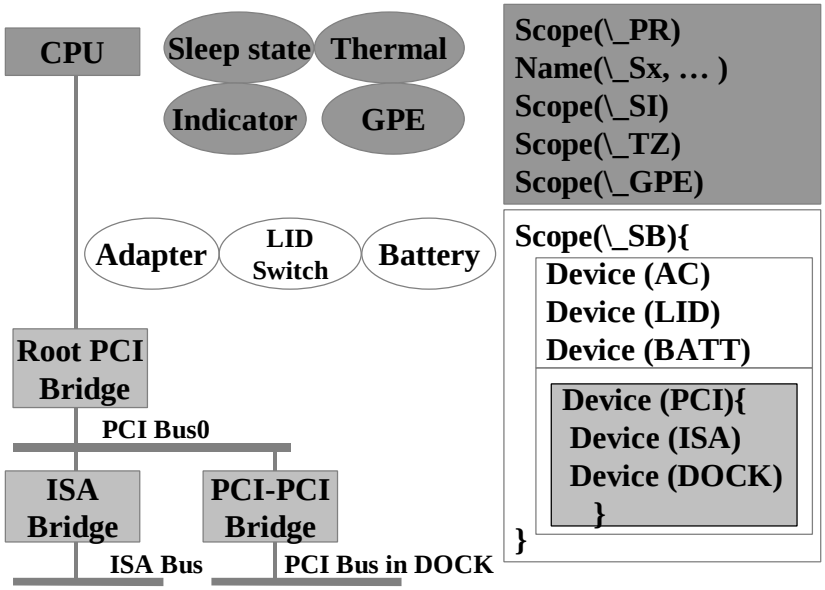


Figure 2. The Name Space Tree

The figure illuminates how the namespace relates to the actual ACPI objects. For example, the CPU is represented by Scope(_PR). The sleep state is represented by Name(_Sx, …). The Thermal state is represented by Scope(_TZ), and the General Purpose Event is represented by Scope(/_GPE).

2.1.1 Definition Block

The Definition Block contains the header information for the ASL assembler.

```
DefinitionBlock (
      "mobile.aml",          // Output file name
      "DSDT",                      // Signature
      0x01,                        // DSDT Revision = v1.0
      "OEMxy",               // OEM ID
      "mbsmpl",              // Table ID
      0x19970930                   // OEM Revision
)
```

Note: "DSDT Revision 0x01" above refers to the fact that this system meets the requirements of the ACPI Specification, Revision 1.0. There are no stipulations for the OEM ID, Table ID, and OEM Revision. The above values for these are examples only.

2.2 Enumerating Buses

2.2.1 Root Objects

The following root objects exist in the ACPI system:

- Sleep state object
- GPE event definition
- Thermal object (optional)
- System indicator (optional)

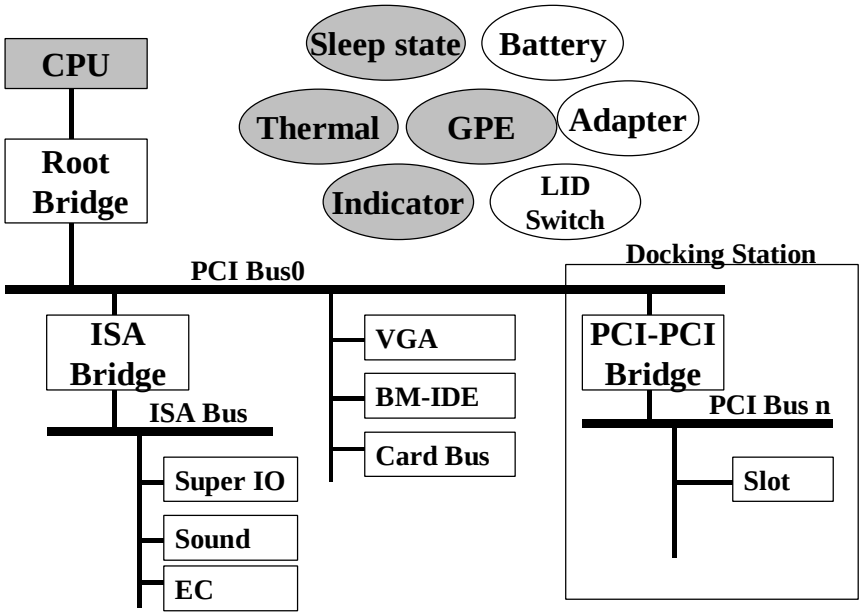


Figure 3. Root Objects

2.2.1.1 Sleeping State Object

This object defines the sleeping state supported by the system. The names of the sleeping states are as follows in ASL:

```
Name (\_S0, Package(){7,        //PM1a_CNT.SLP_TYP
                      7,        //PM1b_CNT.SLP_TYP
                      0}        //Reserved
      )
Name (\_S2, Package() {6,6, 0})
Name (\_S3, Package() {4,4, 0})
Name (\_S5, Package() {0, 0, 0})
```

Note that ACPI requires at least one sleep state, S1, S2, S3, or S4 (for more information about each of the Sx states, see the topic "Definition of Terms" in this *ACPI Implementers' Guide*). Also note that S1 and S4 are not supported by the Mobile Concept machine.

2.2.1.2 Thermal State

The following ASL code defines the thermal zone:

```
Scope(\_TZ) {
      PowerResource(PFAN, 0, 0) {
         Method(_STA) {
       //Returns state of power resources
         }
         Method(_ON)  {…}  // turn on fan
         Method(_OFF) {…}  // turn off fan
      }
      // Create FAN device object
      Device (FAN) {
         Name(_HID,EISAID("PNP0C0B"))                    // PNP ID for FAN
         // list power resource for the fan
         Name(_PR0, Package(){PFAN})
      }
      // create a thermal zone
      ThermalZone (THRM) {
         //Method(_TMP) {…}                    // get current temp
         Method(_AC0) {…}                      // active temp
         Name(_AL0, Package(){FAN})        // fan is act cool dev
         Method(_PSV) {…}                      // passive temp
         Name(_PSL, Package(){\_PR.CPU0})                    // cpu is passive dev
         Method(_CRT) {…}                      // get critical temp
         Method(_SCP, 1) {…}                   // set cooling mode
         Name(_TC1, 2)                              // thermal coefficient
         Name(_TC2, 3)                              // thermal coefficient
         Name(_TSP, 600)                       // sample every 60sec
      }  // end ThermalZone object THRM
    } //end _TZ
```

2.2.1.3 System State

ACPI provides an interface for a variety of simple and icon-style indicators on a system. All indicator controls are in the _SI portion of the namespace.

```
Scope (\_SI) {
       Method(_MSG,1 ){                              // message
             // Arg0 has the number of the message
             Store (ARG0,  \_SB.PCI0.ISA.EC0.NMSG)
       }
       Method(_SST, 1) {                             // sleep level
             // Arg0 has the level of the sleep state
             // going to enter
             Store (ARG0, \_SB.PCI0.ISA.EC0.SLED)
       }
}
```

Note: _SI is optional.

2.2.1.4 General Purpose Event Scope

- GPE.00 is connected to the embedded controller.
- GPE.0B is connected to LID wakeup.
- GPE.0E is connected to Docking.

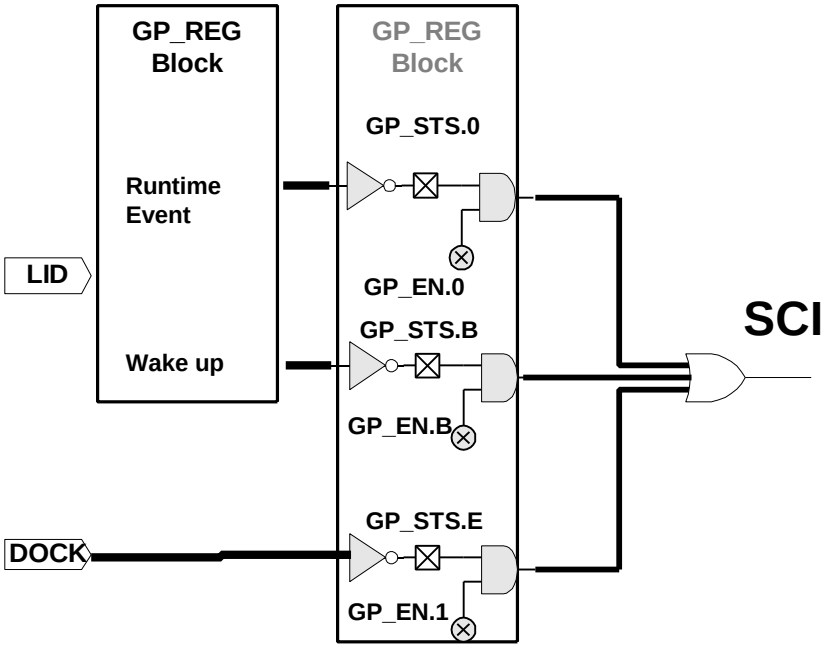


Figure 4. General Purpose Event

The following ASL code defines the GPE event scope.

```
Scope (\_GPE) {
      Method(_L0E){       // for docking
            …
            Notify (\_SB.PCI0.DOCK, 0)
            …
      }    // Docking
}
```

Note: _L00 does not exist. The GPE for the embedded controller must be defined under the EC device.

2.2.2 System Bus Objects (ACPI Devices)

The following ACPI devices are located under the system bus:

- Battery
- AC adapter
- LID switch

```
Scope (\_SB) {
    Device(BAT0){ … }          // Battery
    Device(AC){ … }            // AC Adapter
    Device(LID) { … }          // LID device
```

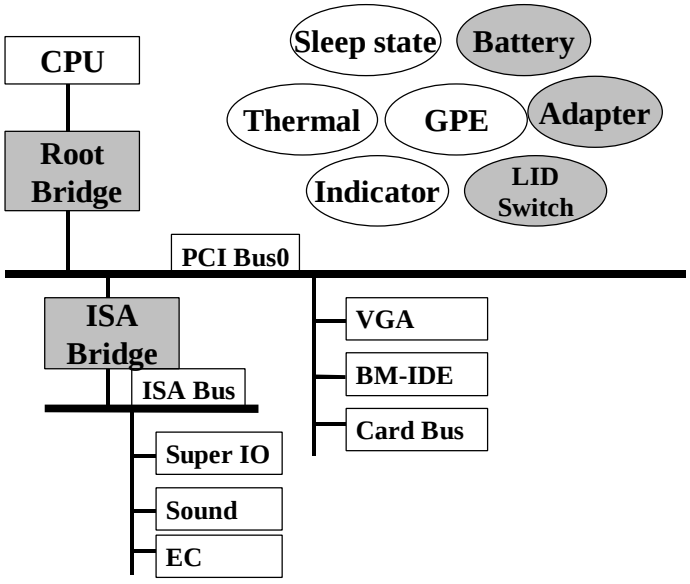


Figure 5. System Devices

2.2.2.1 Root PCI Bridge

The root PCI bridge is the first bridge under the system bus. The PCI and ISA buses are both also under the root PCI bridge, as the following ASL code illustrates:

```
Scope (\_SB) {
    Device(BAT0){ … }              // Battery
    Device(AC){ … }                // AC Adapter
    Device(LID){ … }               // LID device
    ...
    Device (PCI0) {                // Root PCI bridge
        // PCI devices which have a power resource
        …
        Device (ISA) {             // PCI-ISA bridge
            Device (DOCK) {        // PCI-PCI bridge for docking station
```

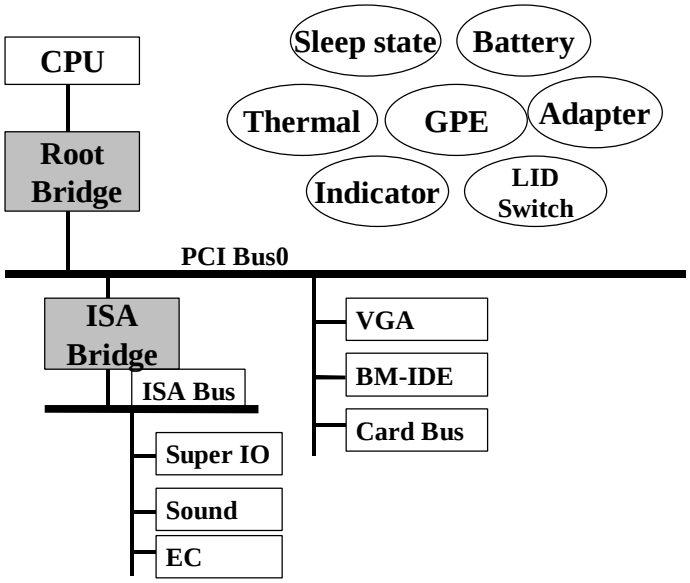


Figure 6. PCI Bus and ISA Bus in ACPI

2.2.2.2 PCI-ISA Bridge for ISA devices

The following ASL code describes the PCI-ISA bridge.

```
Device(PCI0) {                                  // Root PCI bridge
    // Only root bridge require the _HID
    Name (_HID, EISAID("PNP0A03") )
    Name (_ADR, 0x00000000)                     // PCI addr.
    Method (_CRS) { … }                         // report resource
    //PCI Devices
        …
        Device (ISA) {                          // PCI-ISA bridge
            // No need to put the _HID here
            Name (_ADR, 0x000AFFFF)             // PCI addr.
            // ISA Devices
                …
        }
}
```

2.2.2.3 PCI-PCI Bridge for Docking

The following ASL code describes the PCI-ISA bridge. Refer also to the more recent sample ASL code for the Mobile Concept Machine found at the ACPI Website at http://www.teleport.com/~acpi/. The name of the file is mb_smp.asl.

```
Device(PCI0) {                                  // Root PCI bridge
    // Only root bridge require the _HID
    Name (_HID, EISAID("PNP0A03") )
    Name (_ADR, 0x00000000)                     // PCI addr.
    Method (_CRS) {  …  }                       // report resource
        //PCI Devices
            …
        Device (ISA) {                          // PCI-ISA bridge
            // No need to put the _HID here
            Name (_ADR, 0x000AFFFF)             // PCI addr.
            // ISA Devices
                …
        }
    Device(DOCK) {                                  // PCI-PCI bridge
        Name (_HID, EISAID("PNP0A03"))    //_HID for PCI bus
    Name (_ADR,0x0004FFFF)                      // PCI device#,func#
    Method (_UID) {                           //Docking station unique ID
            Return (\_SB.PCI0.ISA.EC0.DCID)
        }
        Method (_STA) {                         //Status of the DOCK
            If (\_SB.PCI0.ISA.EC0.DSTS) {  //if docked
                Return(0x0F)                    //return functioning
                    } else {                    //if undocked
                        Return(0x00)            //return not exist
            }
        }
        Method (_EJ0, 1) {…}                    //supports S0 (hot) undock
        Method (_EJ3, 1) {…}                    //supports S3 undock
        // PCI SLOT IRQ routing
        Name(_PRT, Package(){
                                Package(){0x0001ffff, 0, LNKA, 0},  // Slot 1, INTA
                                Package(){0x0001ffff, 1, LNKB, 0},  // Slot 1, INTB
                                Package(){0x0001ffff, 2, LNKC, 0},  // Slot 1, INTC
                                Package(){0x0001ffff, 3, LNKD, 0},  // Slot 1, INTD
                               }
        ) //end _PRT
    } // end DOCK
}
```

2.3 Enumerating Devices

2.3.1 PCI Devices

If the PCI device has value-added features, such as power control, you must put it in the name space.

2.3.1.1 Card Bus Controller

_INI is a new control method for the cardbus controller. When the operating system boots up, it will call up the _INI control method. _INI disables the PCIC-compatible resource and switches the Cardbus controller to full Cardbus mode.

Note:

- The ACPI operating system does not require the PNP0E03. PNP0E03 should no longer be reported in the name space.
- _INI is only necessary when the system's Cardbus controller boots up in PCIC-compatible mode.

2.3.2 ISA Devices

The ISA bridge generates the ISA bus in the system. All devices under the ISA bridge must be defined here with the proper _HID object.

ISA Devices require the following control methods and objects:

```
_HID    // Device identification object that evaluates to a device's Plug and Play
        // Hardware ID.
_STA    // Power resource object that evaluates to the current on or off state of the
        // Power Resource
_CRS    // Device configuration object that specifies a device's current resource
        // settings, or a control method that generates such an object.
_PRS    // Device configuration object that specifies a device's possible resource
        // settings, or a control method that generates such an object.
_SRS    // Device configuration control method that sets a device's settings.
_DIS // Device configuration control method that disables the device.
```

2.4 Using an Embedded Controller

The embedded controller (EC) represents a special IO defined by the ACPI specification. You can expand the event. The following conditions apply to the EC in the Mobile Concept machine:

- The EC needs PNP0C09 and _GPE.
- A GPE event for the EC is specified in the _GPE object under the EC.
- The Mobile Concept machine uses GPE0 for the EC.
- It is not necessary to include the _L00 control method.
- When EC fires the GPE0, the operating system directly executes the query to the EC.

2.4.1 Expanding Events Through the Embedded Controller

The following figure shows the relationships between the devices that are wired to the embedded controller, the embedded controller queries, and the ACPI-specified General Purpose Register block, which is the source of a system control interrupt.

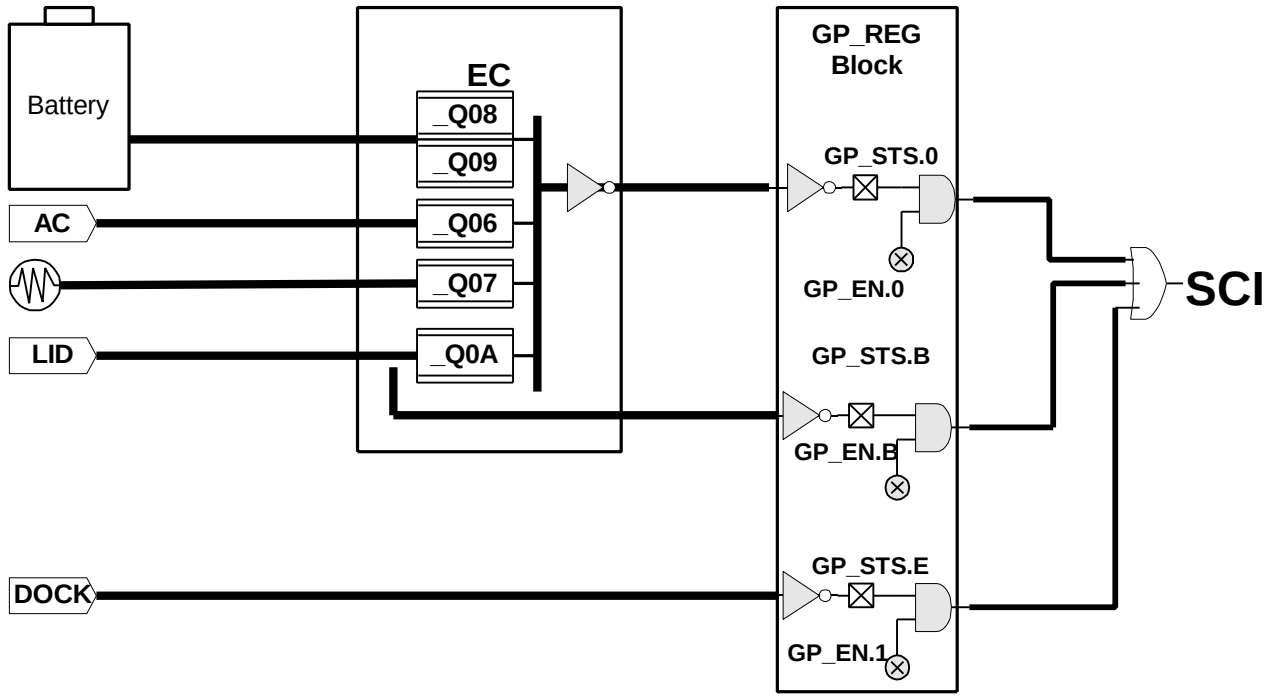


Figure 7. Relationships Between Devices, EC, and GP_REG Block.

2.4.2 Defining the Embedded Controller in ASL

The following ASL code defines the embedded controller.

```
Device (EC0) {
    Name (_HID, EISAID("PNP0C09"))
    Method (_CRS) { … }
    Name (_GPE, 0 )              // EC uses GPE0
    Method(_Q06)  { … }          // Adapter event
    Method(_Q07)  { … }          // Thermal event
    Method(_Q08)  { … }          // Battery event
    Method(_Q0A) { … }           // LID event
```

2.4.3 Example of an EC Event (LID)

- LID open/close generates a GPE.0 runtime event in the EC.
- LID will generate a GPE.B wakeup event when the LID wakeup is enabled.

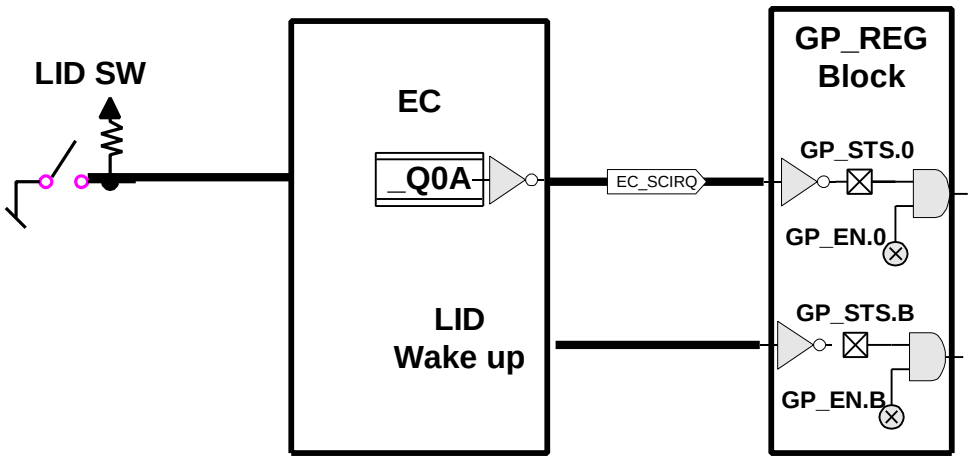


Figure 8. The LID Switch and Relationship to the Embedded Controller and the GP_REG Block

2.4.4 Query Event Processes

The follow figure illustrates the processing of query events for a LID close event.

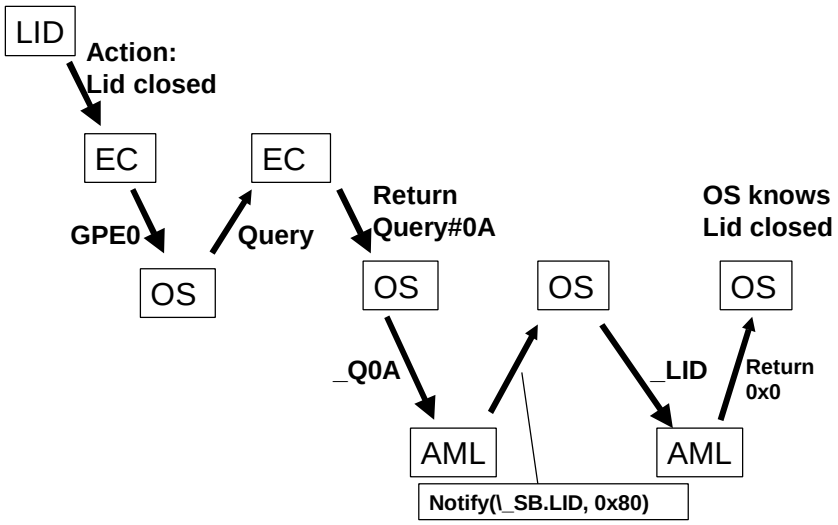


Figure 9. Processing of Query Events for a LID Event

2.4.5 LID Wakeup Event

LID wakeup uses the following objects and control methods:

- _PRW indicates LID wakeup uses GPE.B.
- _PSW enables/disables LID wakeup.

LID will generate a GPE.B event upon opening , when LID wakeup is enabled by _PSW.

The following ASL code defines the LID device:

```
Device (LID) {
      Name (_HID, EISAID("PNP0C0D")
      Method (_LID) {
            Return ( … )                        // Return Open/Close
      }
      Name (_PRW, Package(){
                                    0xB, // use GPE.0B for wakeup
                                    3} )  // Can wake from S3
      Name (_PSW, 1) {
            // Arg0 indicates enable/disable for LID wakeup
            // Needs to set the LID wakeup function enable/disable
      }
}
```

2.5 Docking Events

Docking is covered in detail in Chapter 7 of this Guide.

The Mobile Concept machine uses the following objects in docking:

- _EJ0 indicates the capability of hot docking.

- _EJ3 indicates the docking station supports a warm dock.

When a dock/undock event happens, GPEx fires, and the operating system will invoke _Lxx. _Lxx issues a Notify wake, and indicates that the status of the docking station has changed.

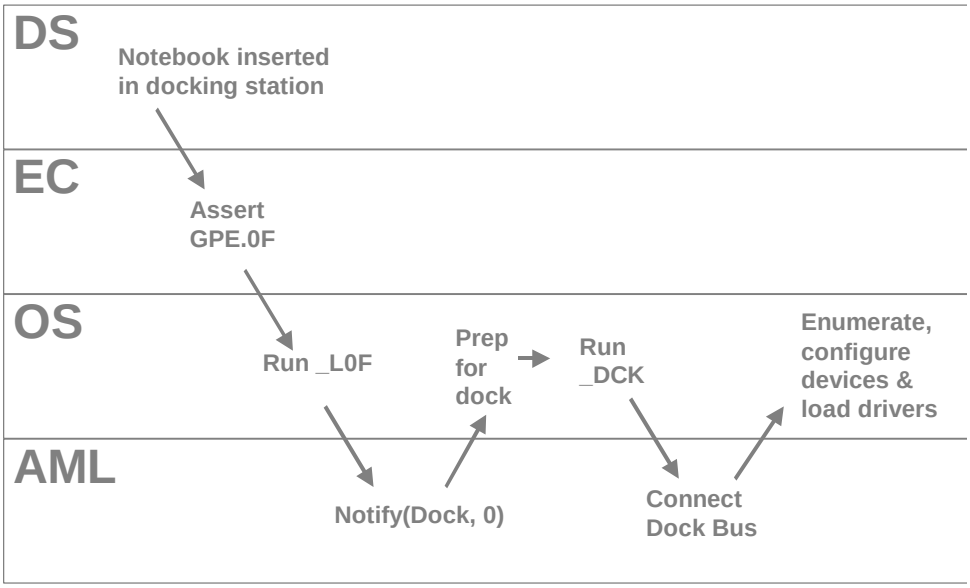


Figure 10. Docking Events (See Chapter 7 for more detail.)

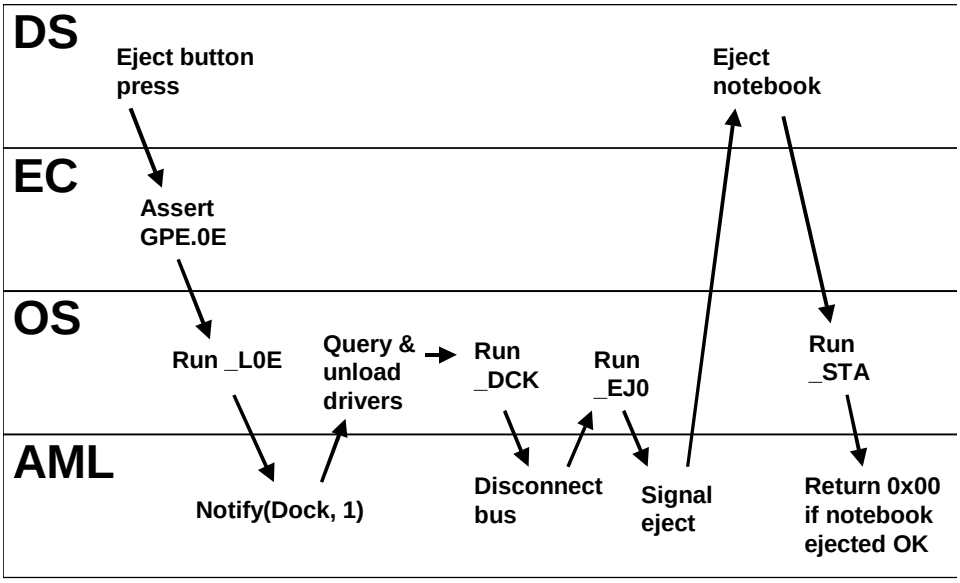


Figure 11. Hot Eject Events (see Chapter 7 for more detail)

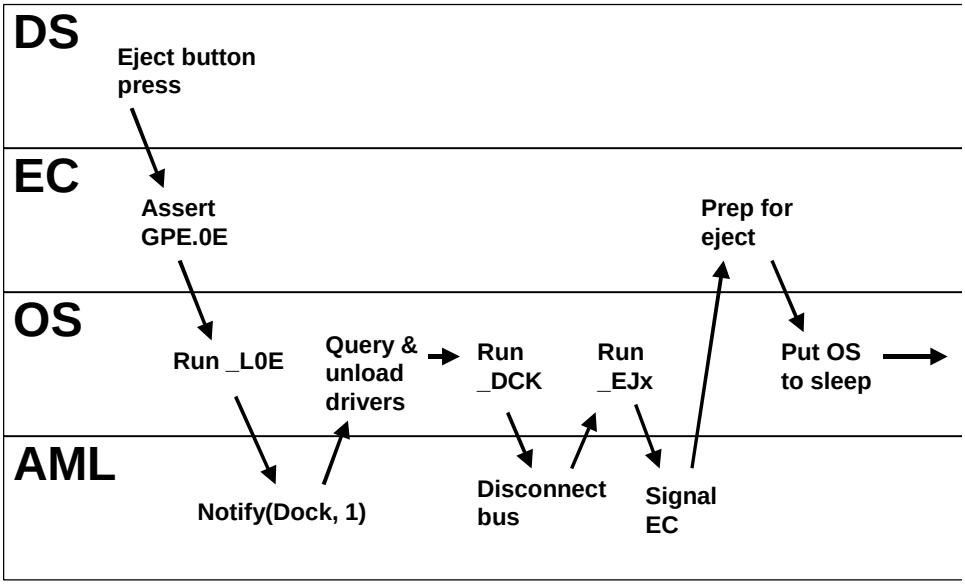


Figure 12. Warm Eject Events (See Chapter 7 for more detail.)

~~1.1.1~~2.5.1 Docking-Related Devices (Joystick)

The following figure illustrates an example of a machine designed with joystick capability through the docking station. On a Mobile Concept machine, the joystick port will only appear in the user interface (UI) when it is docked. When undocked, the port will not appear in the UI.

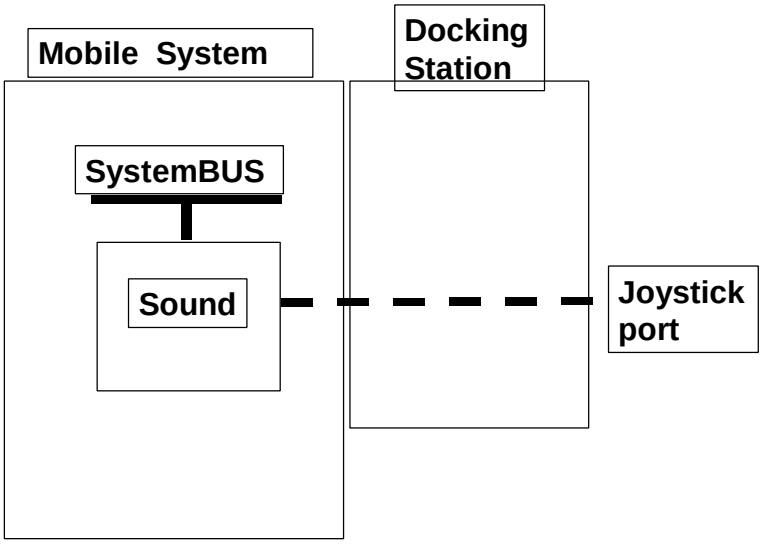


Figure 12. Joystick Docking Configuration

The _EJD control method indicates that the joystick is a docking-dependent device, as the following ASL code illustrates.

```
// Joystick port is on the docking station.
// Joystick will only appear in UI when docked.
//
Device(SND1) {                              // Joystick (Game Port)
      Name(_HID,EISAID("SND0001"))          // example ID for Joystick
      Name(_EJD,"\_SB.PCI0.DOCK")           // means dock-dependent device
      Method(_STA,0) {                      // Status of the Joystick
            If (\_SB.PCI0.ISA.EC0.DSTS) {  // docked and functioning
                  Return (0x0F)             // return show UI
                        } else {            //When undocked and not shown in
                  Return (0x03)             //return remove UI
      }
      Method (_CRS) {                       // Current Resources
            //Prepare the current resource of UART
            //Name (BUFF, buffer(size){data})
            //Return (BUFF)
      }
      Method (_PRS) {                       // Possible Resources
            //Prepare the current resource of UART PORT
            //Name (BUFF, buffer(size){data})
            //Return (BUFF)
      }
      Method (_SRS,1 ) {                    // Set Resources
            // ARG0 = PnP Resource String to Set
            //Control of setting resource is placed here
      }
      Method (_DIS,0 ) {                    // Disable Resources
            //Control of setting resource is placed here
      }
} // end SND1
```

3. ACPI Event ImpGuide

3.1 Connecting a Power Management Event Signal (PME#) in an ACPI System

The Power Management Event signal is defined in the *PCI Bus Power Management Specification, Version 1.0*. (The specification is available at http: //www.pcisig.com.) It is treated as level-mode wakeup signal in ACPI. The PME# signals from all devices must be wire-OR'd together. It is recommended that you use separate events for each root bus in your configuration. Note that PME cannot be connected to a non-wakeup (runtime) event signal.

There are currently two ways to connect a PME# signal to an ACPI system:

- Connecting a PME# directly to a General Purpose Event (GPE)
- Connecting a PME# to the Embedded Controller (EC) using level triggering

3.2 Connecting PME# Signal Directly to a Dedicated General Purpose Event

In this implementation, the PME# signal from the PCI bus is connected directly to a GPE pin on the core chipset. Thus a GPE is dedicated to use for PME#.

This is the preferred implementation as well as the one intended by the *PCI Bus Power Management Specification, Version 1.0*. (The specification is available at http: //www.pcisig.com). In your ASL code, the following will be required under the PCI bus:

Requirements:

- Every root PCI bus device object must have a _PRW object that will list correctly the power resources and system states necessary to detect a PME# signal. This _PRW indicates which GPE is dedicated to PME#
- You will need to associate a PME# signal with its dedicated GPE.
- Remember: Wake events and runtime events cannot be shared on the same GPE! This implementation works best if PME is not shared with anything.

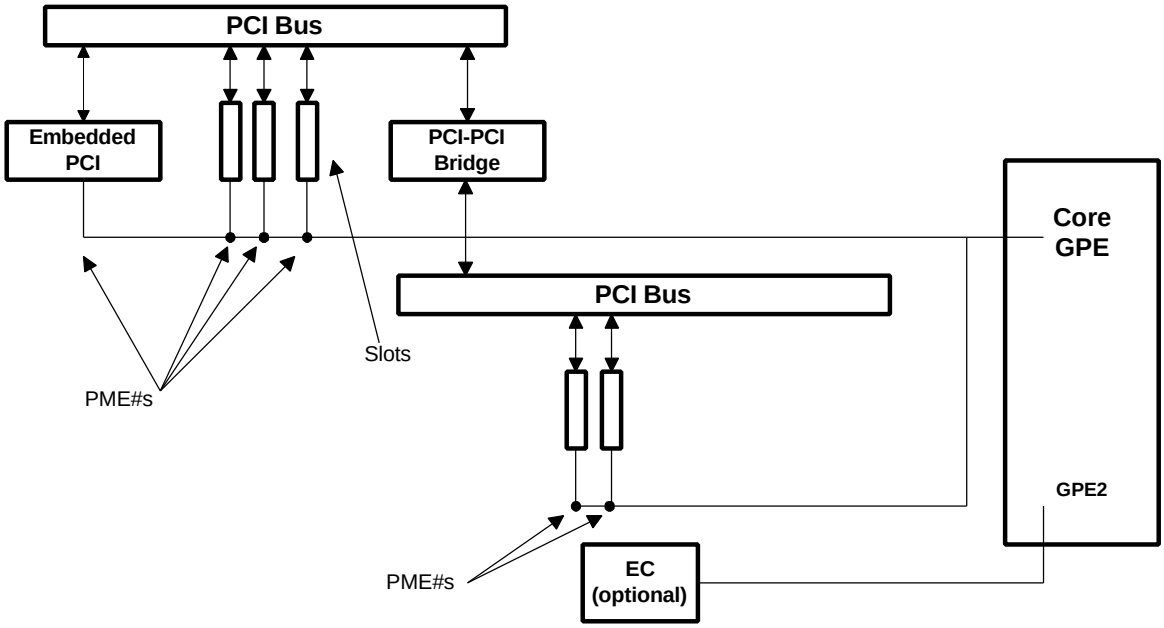


Figure 1. Connecting PME# Signals Directly to a Dedicated GPE

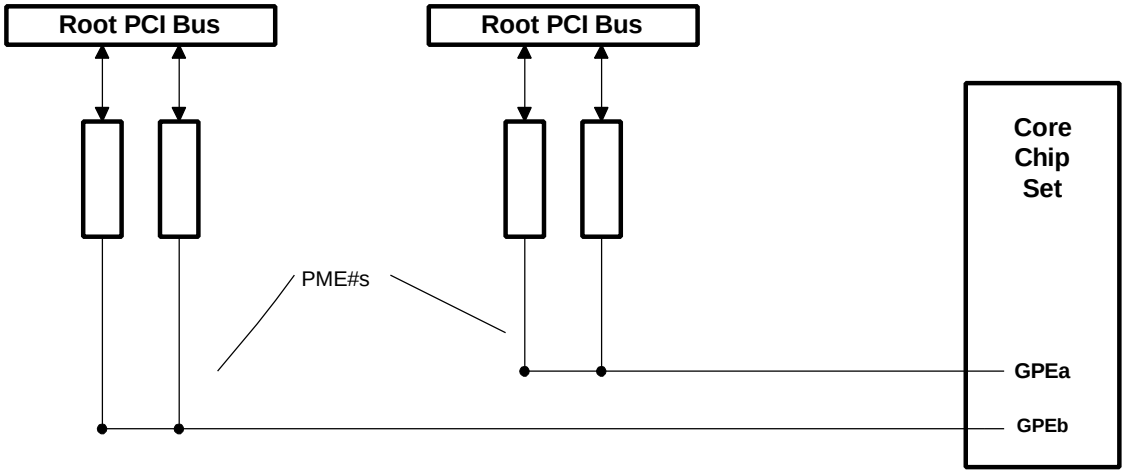


Figure 2. Connecting PME# Signals with Peer Root Buses

3.3 Connecting a PME# Signal to the Embedded Controller Using Level Triggering

In this implementation, the PME# signal travels into the embedded controller. The EC asserts its wakeup signal when the PME# signal is asserted.

Note: The EC's wakeup signal is not the same GPE that the EC utilizes as part of its EC interface, (that is, the GPE listed under the _EC object). It is dedicated for wake events only. Again, wake events and runtime events must not be shared on the same GPE.

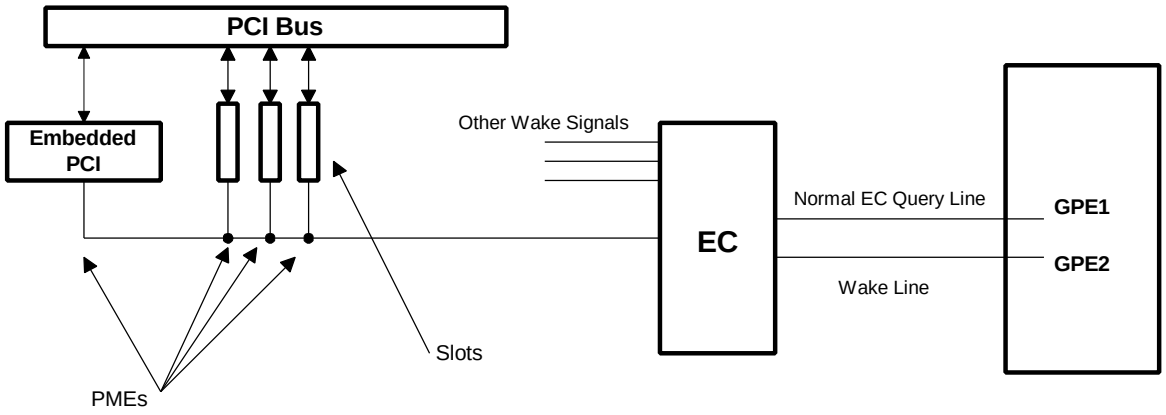


Figure 3. Connecting a PME# Signal to the Embedded Controller Using Level Triggering

Requirements:

- Wake events and runtime events must not be shared on the same GPE.
- You can share PME with other wake events, however you must ensure that each wake event has separate status and enable registers that are controlled by the relevant _PSW control method.
- The GPE must be asserted for the entire duration of the time that the PME# signal is asserted. That is, when the PME# signal is asserted, the GPE is asserted and will not be de-asserted until the PME# signal is first de-asserted.

Additionally, to successfully implement this scenario, the following objects and control methods will be required under the PCI bus:

- Each device (including each root PCI bus) that shares the GPE for wakeup must have a _PRW object that correctly lists the proper resources and system state required to detect PME# signal and that points to the wake GPE from the EC.
- Each device (including each root PCI bus) that shares the GPE for wakeup must have a _PSW control method to enable wakeup. The _PSW must access the EC to inform it that the PME# wakeup signal should be enabled.

3.4 Connecting Runtime Events Using the PIIX4 Chip Sets

There are currently two methods to connect runtime events and wake events to the GPE1, RI, and LID.

- The PIIX4 method
- The PIIX4e method

3.5 PIIX4 Method of Connecting Runtime Events

In this method, runtime events are connected to GPE1 and wake events are connected through the embedded controller to LID.

Note: Be sure to debounce the wake signal en route to the EC. Also note that this implementation does not use Ring Indicate (RI).

3.6 PIIX4e Method of Connecting Runtime Events

In this much more straightforward method, runtime events are connected to GPE1 and wake events are connected to RI.

To sum up,

Event Type	**Method 1**	**Method 2**
Runtime Events	GPE1	GPE1
Wake Events	LID via EC (debounce required)	RI

4. ACPI Embedded Controller ImpGuide

Long battery life is a goal of all mobile platform designers and implementers and the small mobile platform case requires thermal control. This chapter of the *ACPI Implementation Guide* shows how an embedded controller (EC) can be used on a mobile platform to achieve long battery life and thermal control.

4.1 General Use of an EC

An EC is an optional component of ACPI-compliant platforms, but is recommended for ACPI-compliant mobile platforms.

4.1.1 Advantages of Using an EC

Using an EC provides flexible system design because:

- More events can be handled.
- For each event, part of the event processing, or additional event processing, can be done inside the EC.

If an EC is used, it can be either an ACPI-compliant EC or a non-compliant EC. The features of an ACPI-compliant EC are summarized in the topic "EC Architecture and Functionality." If an EC is used that does not comply with the ACPI specification, it is transparent to the OS because no standard interface between the OS and the EC is defined. Events are handled through GPE generic control methods. This chapter of the *ACPI Implementation Guide* describes the architecture and uses of an ACPI-compliant EC.

4.1.2 EC Interfaces

An ACPI-compliant EC typically has three interfaces:

- A keyboard interface, which uses I/O ports 0x60 and 0x64. These I/O port addresses are owned by the OS and are not shared.
- An SMI interface.
- An ACPI (SCI) interface. Separate I/O port addresses are recommended for the SMI and SCI interfaces. If the SCI and SMI interfaces shared I/O ports then the Global Lock protocol must be implemented. For more information about the Global Lock protocol, see section 5.2.6.1 of the *ACPI Specification*.

A block diagram of an EC is shown in Figure 1.

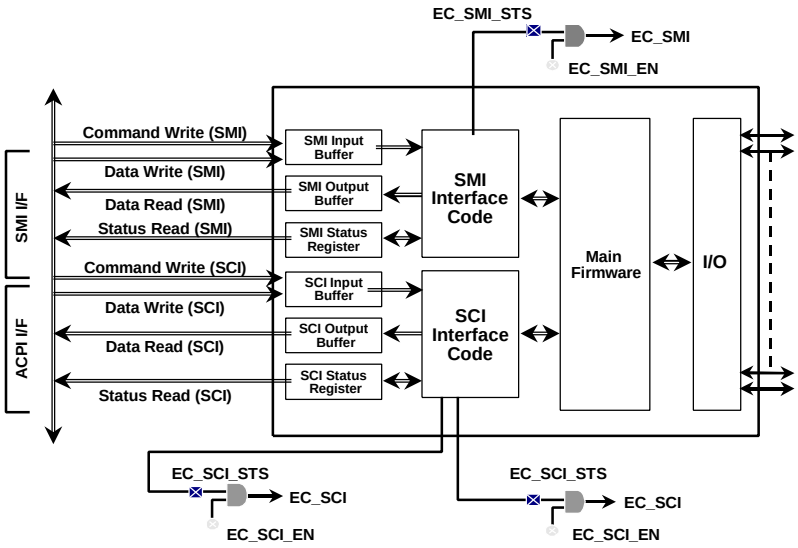


Figure 1. Embedded Controller (EC) Block Diagram

4.1.2.1 The ACPI (SCI) Interface

An ACPI-compliant EC provides a standard interface to the OS (for more information, see section 13 of the *ACPI Specification*. This standard interface is implemented as a 256-byte I/O space in the EC; the OS accesses the EC's SCI interface directly.

The OS acquires event requests through a query mechanism. that involves using an _GPE control method in ASL code. Control methods must not be associated with this GPE, the OS understands how to process this event.

4.2 Connecting the EC to the Chipset

This section presents a set of rules for connecting the event signals that enter an ACPI-compliant EC to the pins on the chipset.

All run-time event signals received by the EC are connected to one GPE pin on the chipset and all wake events are connected to a different GPE pin the chipset.

- *Run-time events* are events that occur while the system is in the S0 state.
- *Wake-up events* are events that occur while the system is in one of the sleeping states. For example, a power-button event is a wake-up event while the system is in the sleeping state.

Figure 2 shows a wrong way to connect run-time and wake-up event signals from an EC to the chipset; it is an error to connect both a run-time and a wake-up event signal to one GPE pin on the chipset.

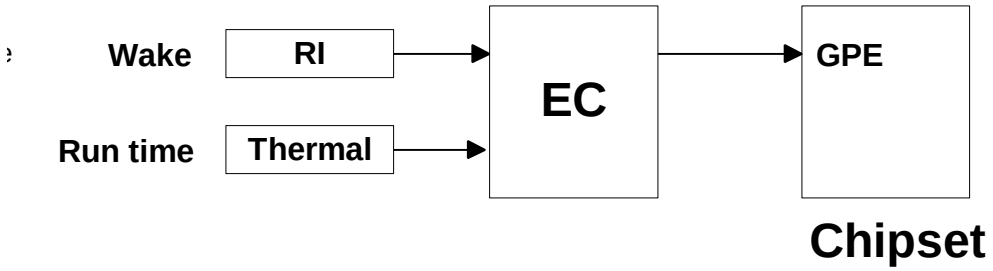


Figure 2. Wrong Way to Connect Event Signals to the EC and Chipset

Figure 3 shows a connection diagram that correctly connects run-time and wake-up event signals from the EC to the chipset. The EC is connected to the chipset with two GPE pins and the EC arbitrates the wake-up and run-time event signals it receives, routing all of the run-time events to one GPE pin and all the wake-up events to the other GPE pin on the chipset.

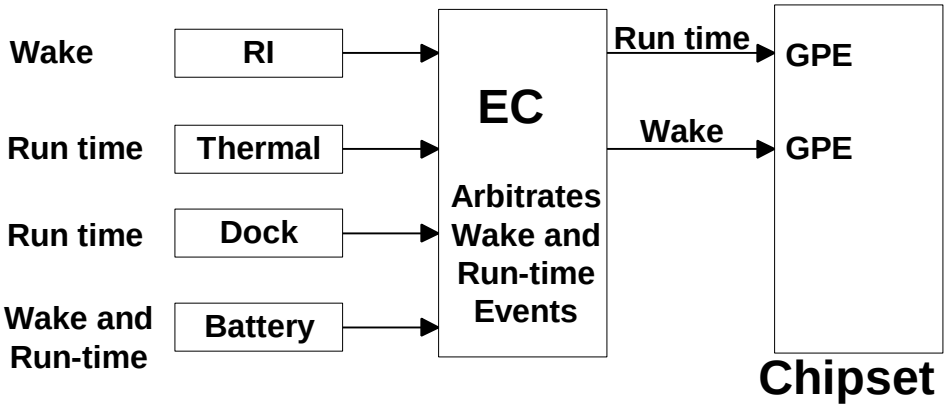


Figure 3. Correct Way to Connect Event Signals to the EC and Chipset

4.3 Writing ASL Code for Wake-Up Events

Examples of devices capable of generating wake-up events, or *wake-up devices*, are the power button, the PCI bus (through PME events), batteries, and modems. In your ASL code, the Device objects that describe wake-up devices must contain _PRW objects.

A _PRW object

- Defines the GPE pin that generates a wake-up event.
- Defines the sleeping states from which the device can wake the system.

The OS determines run-time and wake-up events by using _PRW objects.

The following example ASL code shows a set of objects that define run-time and wake-up events. Note that in the EC device object a _GPE object is used to define the EC GPE; do not use _Lxx, _Exx, or _PRW objects for that purpose.

```
Scope(\_GPE) {
        .
        .
        .
        Method (_L09) {         // Bit 9 of GPE is for run-time events
                .
                .
                .
} //end \_GPE scope
.
.
.
Scope(\_SB) {
        .
        .
        .
        Device(XYZ0) {          // Device object for wake-up device
                .
                .
                .
                Name (_PRW, Package(2){
                                                5,      //Bit 5 of GPE is for wake-up events
                                                4       //Lowest sleep state to wake from is S4
                ) //end of _PRW object
                .
                .
                .
        } //end of XYZ0 device object
        .
        .
        .
        Device(EC0) {           // Device object for Embedded Controller                                .
                .
                .
                Name((_GPE, 0x08)               //EC uses GPE bit 8
                .
                .
                .
        } //end of EC device object
        .
        .
        .

} //end of \_SB scope
.
.
.
```

Note that the OS disables run-time events before going to sleep. The steps the OS takes to put the system to sleep are:

1. OS decides to sleep.
2. OS invokes the _PTS control method.
        .
        . (an indeterminate amount of time goes by)
        .
3. OS disables run-time events.
4. OS enables wake-up events
        . (OS enters “wake mode”)
        .
        .
5. System enters the sleep state.

4.4 Prolonging Battery Life

Mobile platform battery life can be prolonged by

- Power plane control.
- Battery control (for example, predicting remaining time and doing intelligent charging).

4.4.1 Smart Battery Subsystem Architecture and How It Is Used by the OS

An ACPI-compatible smart battery subsystem consists of:

- Battery, selector, and charger residing on SMBus.
- SMBus host controller interface emulated in the ACPI-compliant EC interface.

The OS communicates with the smart battery subsystem components through the EC interface (for more information, see section 13.9 of the *Revision 1.0 ACPI Specification*). The SMBus register set occupies 40 bytes of the EC's I/O space.

Note: *The Smart Battery Data Specification, Revision 1.0* is available at http://www.sbs-forum.org/specs.htm

A block diagram of a smart battery subsystem is shown in Figure 5.

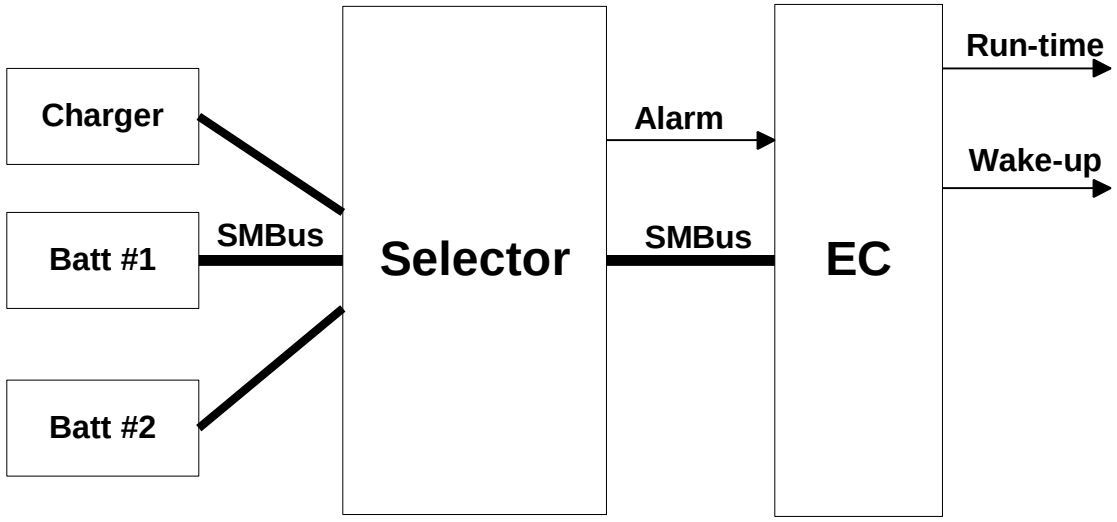


Figure 4. Smart Battery Subsystem Architecture

The battery (or batteries), charger, and selector must notify the OS of state changes by using either alarm messages or a hardware signal. The EC must translate this notification and pass it to the OS through the ACPI EC interface, and generate an SCI.

The OS automatically accesses charger and battery information. No control methods are necessary and specified battery data accuracy is guaranteed.

Examples of messages that need OS attention are:

- User-low or user-critical messages; in this case, the OS transitions the system to the user-specified sleep state.
- Battery-critical message; in this case, the OS exercises emergency shutdown.
- Smart battery subsystem state changes, such as AC presence or battery insertion and removal; in this case, the OS may use the information to adjust the user interface (UI) and/or thermal control policy.

4.4.2 Writing ASL Code to Describe a Smart Battery Subsystem

The following example ASL code describes a smart battery subsystem.

```
Device(EC0) {                   //ACPI Embedded Controller
    Name(_HID, EISAID("PNP0C09"))
    Name(_GPE, 0x09)                    //EC uses GPE bit 9 for runtime event
    Method(_STA) {
        Return(0x0F)                    //Present, enabled
    }
    Name(_CRS, Buffer() {                   //IO port 0x62/0x66
                                    })
    Device(SBS0) {              //SMBus host controller
        Name(_HID, "ACPI0001")
        Name(_EC, 0x0001)               //SBS registers occupy EC IO space
                                            //0x00-0x3F, query number is 1
        Device(SBS0) {      //Smart battery subsystem
            Name(_HID,"ACPI0002")
            Name(_SBS, 0x02)            //SBS supports 2 batteries
        }
    }
}
```

4.5 Thermal Control

The small case of a mobile platform requires thermal control and the EC can be used for this.

4.5.1 ACPI Thermal Control Model for Mobile Systems

The general ACPI thermal control model is

- The OS makes cooling decisions, based on user choices for performance and noise levels and the power sources on the platform.
- Multiple temperature trip points can be defined for a platform and that is recommended.

Passive cooling, active cooling, and critical trip points can be defined.

- One passive trip point can be defined around which the CPU does closed-loop thermal regulation, using CPU throttling.
- From 1 to 10 different active trip points can be defined

4.5.2 How Thermal Trip Points are Used by the OS

A thermal SCI is generated when a trip point is crossed and the OS responds to that. For example, the OS can change the cooling policy by invoking the _SCP control method. The figure below shows a graphical representation of two different cooling policies the OS could choose from.

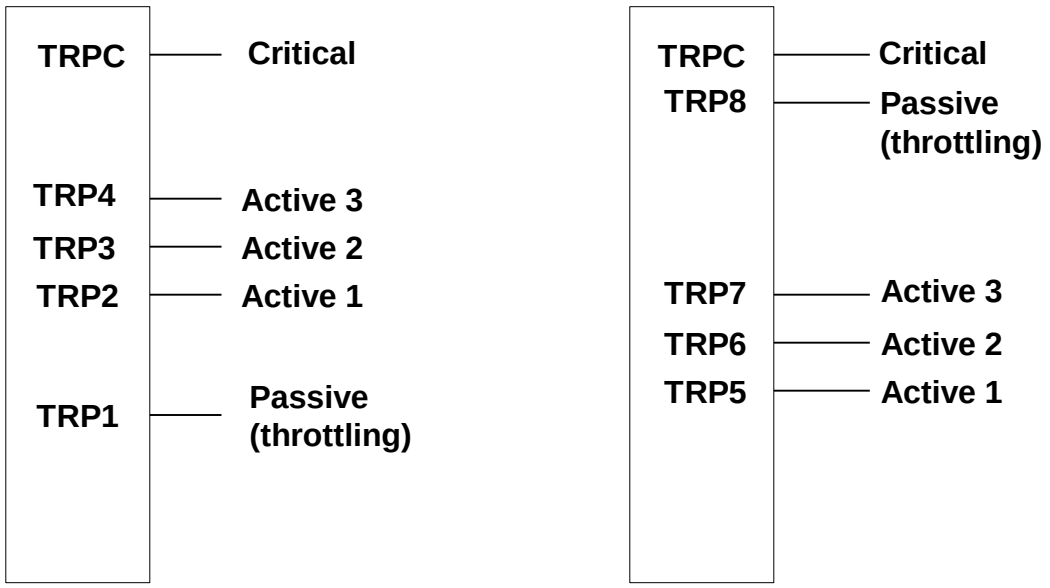


Figure 5. Two Example Thermal Policies with Multiple Trip Points

4.5.3 Thermal Sensor Implementations

Some thermal sensor implementation options are:

- Put the thermal sensor behind the EC.
- Connect the thermal sensor to the chipset using the SMBus; this can be done for a multiple-trip point sensor or for a single-trip point sensor.

4.5.3.1 Putting the Thermal Sensor Behind the EC

The following block diagram illustrates placing the thermal sensor(s) behind the EC.

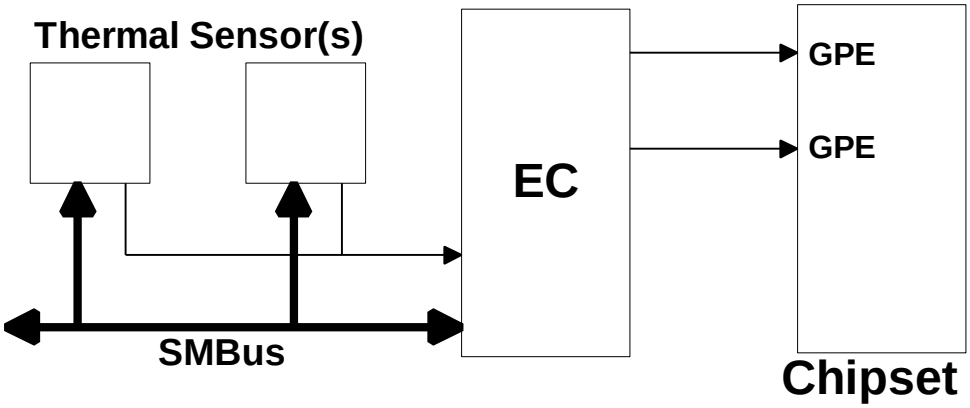


Figure 6. Thermal Sensor(s) Behind the EC

This design option offers a lot of flexibility: multiple temperature trip points can be emulated and you can use a simple thermal sensor.

The critical trip point can be defined by the EC or the thermal sensor.

- The EC should have a watch dog timer.
- The EC can shut down the system, so this works even when the OS crashes.

4.5.3.2 Connecting the Thermal Sensor to the Chipset

The following block diagram illustrates connecting the thermal sensor to the chipset.

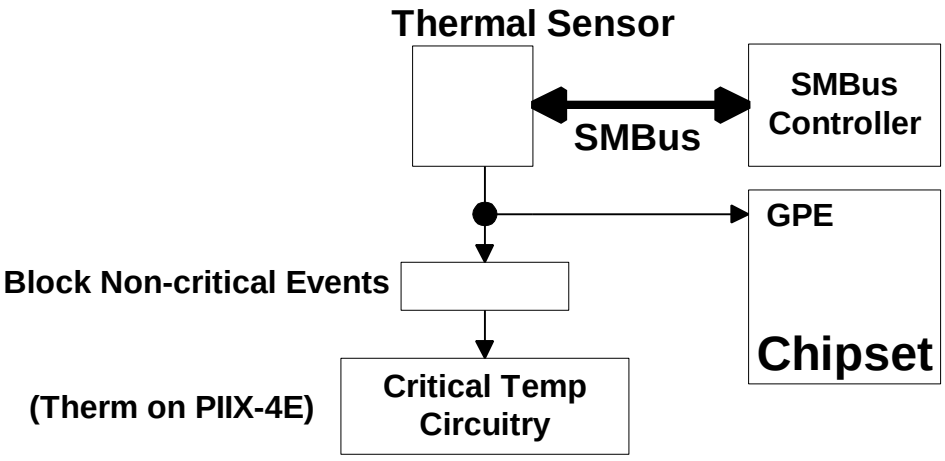


Figure 7. Connecting the Thermal Sensor to the Chipset

In this design, the temperature trip points are defined by configuring the thermal sensor. The thermal sensor should support multiple temperature trip points to meet the requirements of a mobile platform design.

5. Installing and Using the AMLI Debugger

This chapter describes how to install and use the AML debugger. These procedures are somewhat different under Windows 98 and under NT.

5.1 Debugging ACPI Systems under Windows 98

The following executable files are required for running the AMLI debugger under Windows 98:

- WDEB386.EXE; the Windows 9x kernel debugger.
- RUNWDEB.BAT; a batch file that invokes the kernel debugger and sets up various parameters for it. Do NOT invoke the debugger directly. Run RUNWDEB.BAT to invoke the kernel debugger.

5.1.1 Installing the Debugger

The AML debugger works through the operating system kernel debugger. To use the AML debugger, you must first set up the OS kernel debugger. This requires:

- A test machine with WDEB386.EXE and RUNWDEB.BAT
- A debug terminal machine connected to the test machine via a null modem cable, and running a terminal program.

The procedure for setting up the test machine is:

1. Be sure that RUNWDEB.BAT and WDEB386.EXE are in the subdirectory c:\windows\system.
2. Edit RUNWDEB.BAT if necessary.

   The RUNWDEB.BAT file has been configured to support most environments. Depending on the COM port you plan to send debugging data out to you may need to change the /C:1 portion of the command line to /C:2 reflect COM2.
3. Update your AUTOEXEC.BAT file on the test machine to include the following lines:

```
REM Goto End
cd\windows\system
pause
runwdeb
:End
```

   This way if you want to disable the debugger through multiple boots you can take the REM prefix off the Goto statement and avoid starting the debugger at all. The pause command makes last minute changes much easier.
4. You will now need to insert a key in the registry to break into the AML debugger so that you can set breakpoints before encountering the problematic code.

The procedure for inserting the key in the registry is:

1. Use a Registry Editor tool to locate the System key in the Registry of the test machine platform. This key is

   HKEY_LOCAL_MACHINE\System\Current\ControlSet\Services\ACPI~~Class\System~~Parameters
2. ~~Under the System key in the Registry there are two or three dozen keys, each with a unique number (0000, 0001, ... 0022, ...). Look through these keys until you find one for which the DriverDesc string value is "Advanced Configuration & Power Interface 'ACPI' BIOS"; it could be any of the two or three dozen keys.~~
3. In the above key ~~you just found~~, use the New Binary Value feature of your Registry Editor to add a binary value named "AMLIInitFlags" with one of the following values:
   - 01 00 00 00 if you would like to break into the AML debugger after the interpreter initialization is completed.

- 02 00 00 00 if you would like to break into the AML debugger after any definition block is loaded.
- 03 00 00 00 if you want to break into the AML debugger in both cases.

5.1.2 Running the Debugger

For performance reasons, the AML debugger is not enabled by default. To use the AML debugger, you must first enable it.

Once you have installed the components, modified the registry and rebooted the system you should see the following print on the test system:

```
Microsoft (R) Windows 4.0 Kernel Debugger Version 4.0.6 xx/xx/97  xx:xx:xx
Copyright (C) Microsoft Corp. 1990-1997.  All Rights Reserved
```

On the screen running the terminal program you should see:

```
Kernel Debugger  Version 4.0.6 02/24/97 23:52:11  [80386]
```

If you have set up the "AMLIInitFlags" in the registry as mentioned above, you will stop in the AML debugger when the interpreter has completed initialization or loading a definition block. At this point, you can type a "?" to get additional information on the AML debugger options.

If you are at the ## prompt (kernel debugger prompt) you can type in ..ACPI to get back into the AML debugger.

5.2 Debugging ACPI Systems under Windows NT 5.0

Instructions for using the AMLI debugger under NT are:

1. Install the debug (checked) version of acpi.sys in the subdirectory

   ```
   %windir%\system32\drivers
   ```

2. Start the kernel debugger with the -b option, which enables an initial breakpoint.
3. At the kd> prompt, type

   ```
   ed acpi!AcpiDebugPrintLevel 1000
   ```

4. At the kd> prompt, type

   ```
   g
   ```

5. You can set trace on, or set a breakpoint, or whatever you want to do when you see the following prompt:

   ```
   AMLI(? for help)->
   ```

6. Once you have set trace on, set a breakpoint, or whatever, to start running and see your trace of breakpoint, type

   ```
   g
   ```

Under Windows 98, you can break into the Kernel debugger any time by pressing Ctrl+C; then enter the AMLI debugger by typing the command

..acpi

Under Windows NT, the "..acpi" command is not available. You can either

- Set an initial breakpoint (as described above in the topic "Debugging ACPI Systems Under Windows NT") and enter the AMLI debugger at that breakpoint. One of the things you can do at that breakpoint is to set breakpoints at other places in your code; when you restart your code, you can enter the AMLI debugger when you reach one of those breakpoints.
- Use ASL Breakpoint statements to set breakpoints at places where you want to enter the AMLI debugger.

.

6. ACPI Mobile Concept Machine

This section describes a conceptual ACPI mobile personal computer. The primary differences a typical desktop system are that this mobile concept machine:

- Has an embedded controller that monitors the CPU temperature every few degrees.
- Has swappable drive bays.
- Uses a battery for a power source.

The mobile concept machine docking station has PCI slots plus a port for a joystick.

6.1 Mobile Concept Machine Block Diagram

A block diagram of the concept mobile platform is shown in Figure 1. A prominent component of the mobile platform block diagram is the embedded controller (in the upper-left part of the figure).

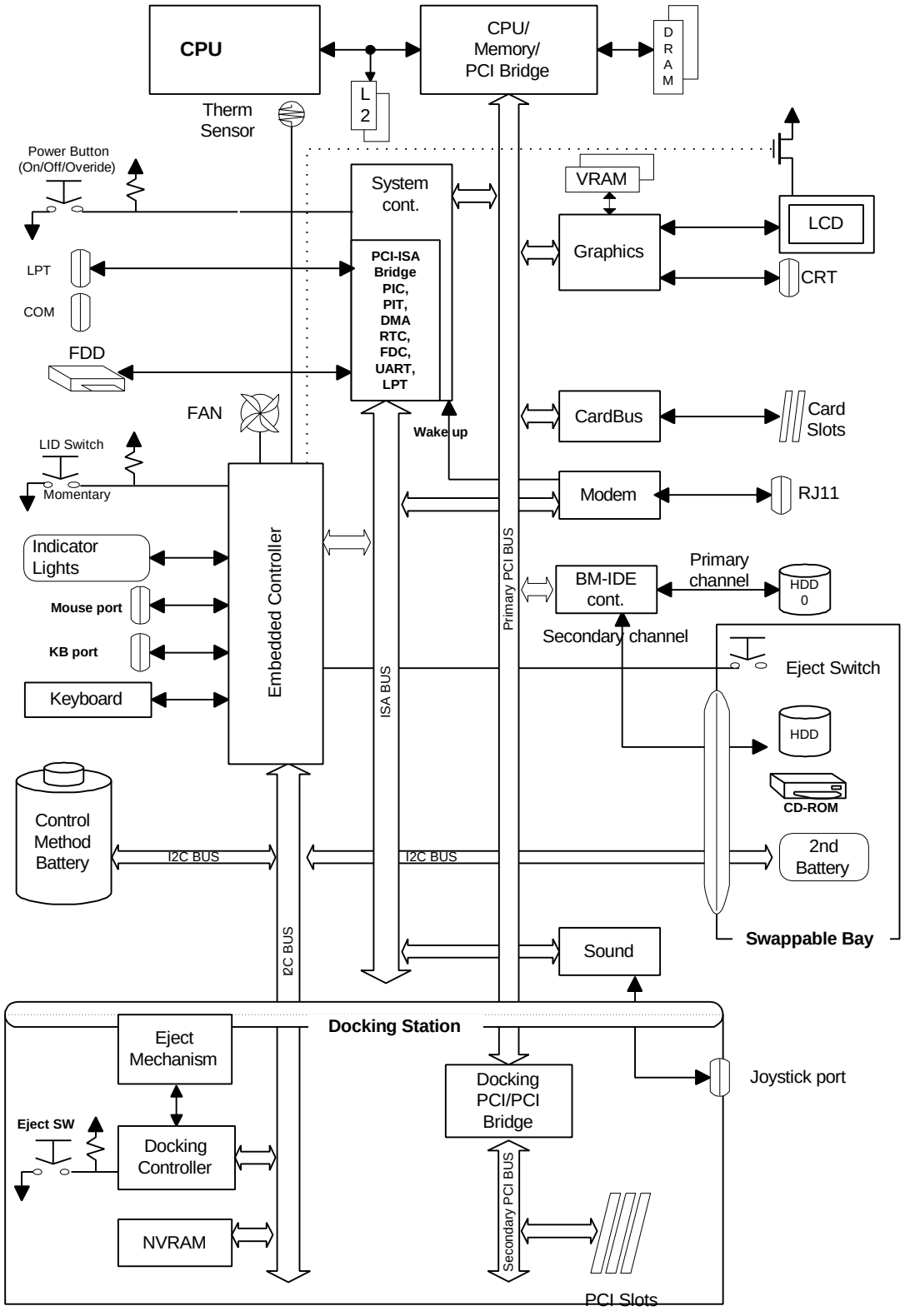

CPU
CPU/
Memory/
PCI Bridge
DRAM
L2
Therm
Sensor
Power Button
(On/Off/Overide)
System
cont.
VRAM
LCD
Graphics
CRT
LPT
COM
PCI-ISA
Bridge
PIC,
PIT,
DMA
RTC,
FDC,
UART,
LPT
FDD
FAN
Wake up
CardBus
Card
Slots
LID Switch
Momentary
Modem
RJ11
Indicator
Lights
Mouse port
KB port
Keyboard
Embedded Controller
ISA BUS
Primary PCI BUS
BM-IDE
cont.
Primary
channel
HDD
0
Secondary channel
Eject Switch
HDD
CD-ROM
Control
Method
Battery
I2C BUS
I2C BUS
2nd
Battery
Swappable Bay
Sound
Docking Station
Eject
Mechanism
Docking
PCI/PCI
Bridge
Joystick port
Eject SW
Docking
Controller
NVRAM
Secondary PCI BUS
PCI Slots

Figure 1. Mobile Concept Machine Hardware Block Diagram

6.2 Devices Used on the Mobile Concept Machine

Prominent devices used on the mobile concept machine are listed in the following table. In some cases, you will have to obtain the Data Sheet for a device to fully understand the ASL methods that define resources for that device.

Table 1. Mobile Concept Machine Devices

Device	Description
Chipset	Up to the OEM
Embedded Controller	Up to the OEM
Video	PCI-based Video device with one power plane.
Modem	Standard modem chip set. Ring Indicate pulled out separately and fed to the RI# wake up input on the chip set.
Control Method Battery	Uses I^2C using control methods.
Audio	Joystick only appears when docked in this concept machine.
2 IDE channel	Primary for the internal HDD, secondary for the bay device.

6.3 Data and Address Bus Structure

This section focuses on the static objects in the ACPI name space for the Mobile concept machine.

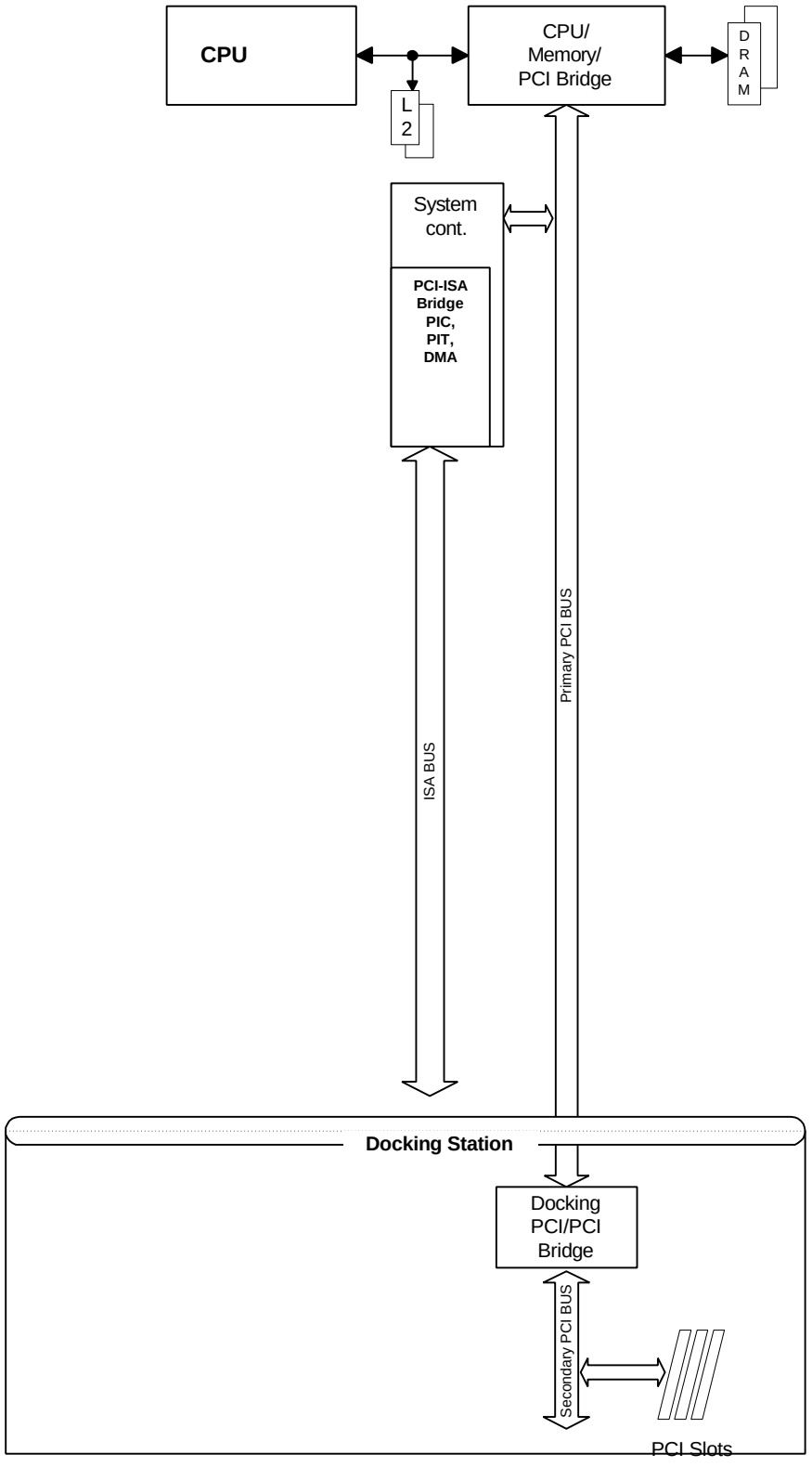
6.3.1 Encoding the Data and Address Bus Structure in ASL

This section shows a phased encoding of the data and address bus structural spine in ASL. The bare bones (data and address bus) structure of the mobile concept machine is made up of the following four scopes:

```
\_SB
    PCI0                    //PCI root bridge device
        _HID                    //PnP ID for PCI bus (used on root bus only)
        _ADR                    //Device address of PCI bus
        _CRS                    //Reports PCI bus number 0 (used on root bus only)
        ISA                     //PCI-ISA bridge
            _ADR                //PCI-ISA bridge address on the PCI bus
        DOCK                    //PCI-PCI bridge (PCI Bus 1 on docking station)

            _ADR                //Device address of PCI bus
            _UID                //Docking station unique ID
```

This ACPI name space model corresponds to the following physical structures from the mobile concept machine block diagram.


CPU
CPU/
Memory/
PCI Bridge
D
R
A
M
L
2
System
cont.
PCI-ISA
Bridge
PIC,
PIT,
DMA
ISA BUS
Primary PCI BUS
Docking Station
Docking
PCI/PCI
Bridge
Secondary PCI BUS
PCI Slots

6.3.1.1 Filling in the Root PCI Bus Scope with Static Device Objects

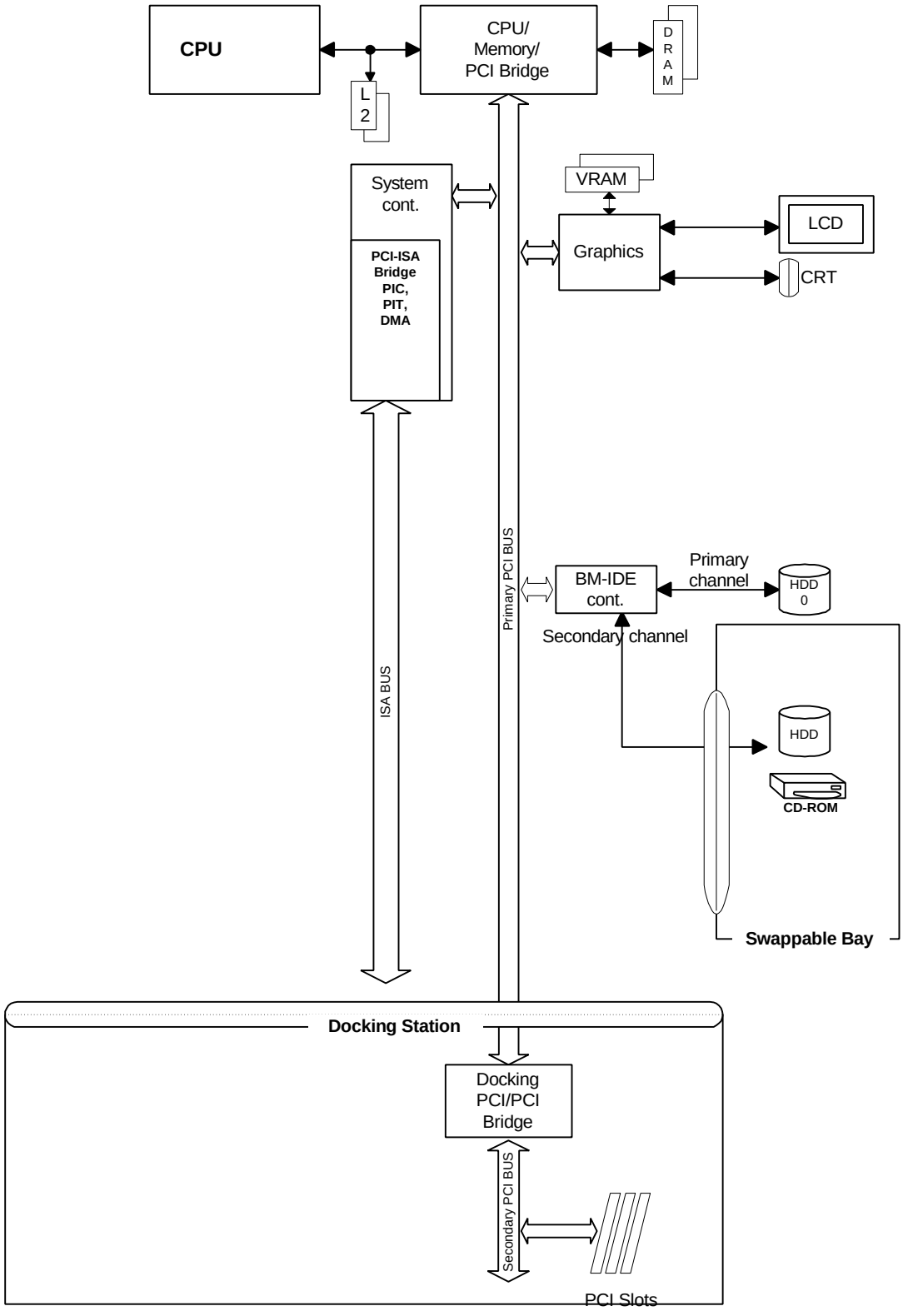
The following ACPI name space model shows the bare bones model with the device objects that hang off the PCI root bus and their configuration objects added to the bare bones objects (the bare bones objects are bolded). The only device hanging off the PCI root bus is video.

```
\_SB
    PCI0                    //PCI root bridge (Host PCI bridge)
        _HID                //PCI Bus ID (used on root bus only)
        _ADR                //address of PCI device
        _CRS                //report resources under PCI root bridge
        IDE1            //IDE channel
_           _ADR                    //Indicates address of the channel on the PCI bus
_           _GTM                    //Control method to get current IDE channel settings
_           _STM                    //Control method to set current IDE channel settings
            DRV1                    //Drive 0
                _GTF                //Control method to get task file
                _ADR                //Master Drive
            DRV2                    //Drive 1
                _GTF                //Control method to get task file
                _ADR                //Slave Drive
        BAY                         //Second IDE channel
            _ADR                    //Indicates address of the channel on the PCI bus
            _LCK                    //means ejectable
_           _GTM                    //Control method to get current IDE channel settings
_           _STM                    //Control method to set current IDE channel settings
            DRV1                    //Drive 0
                _GTF                //Control method to get task file
                _ADR                //Master Drive
            DRV2                    //Drive 1
                _GTF                //Control method to get task file
                _ADR                //Slave Drive
            _
        VID0                //VIDEO Device (PCI)
            _ADR            //address of PCI device
        ISA                 //PCI-ISA Bridge
            _ADR            //Device address of PCI-ISA bridge on PCI bus
        DOCK                //PCI Bus 1 of DOCKING STATION (PCI-PCI bridge)

            _ADR            //Device address of the PCI bus
            _UID            //Docking station Serial #
            _BDN            //Docking Station ID
            _DCK            //Isolation and Remove isolation from connector
            _STA            //Status of docking bridge
```

This name space model corresponds to the elements of the mobile concept machine block diagram shown in the following illustration. An important thing to note is that if you match the following illustration with the block diagram at the beginning of this section, the block diagram at the beginning of the section shows a CardBus device attached to the PCI bus – yet that device is not represented by an object in the ACPI name space. That is because PCI devices that do not have any value-added features from the OEM are not modeled in ACPI name space; the CardBus device on the mobile concept machine is an example of this. The Card Bus device is enumerated, configured, and power-managed by its native bus driver (the PCI bus driver).

In contrast, the VID0 device attached to the PCI bus is represented by an object in the ACPI name space is that the OEM has added a value-added power saving feature to the graphics subsystem.


CPU
CPU/
Memory/
PCI Bridge
D
R
A
M
L
2
System
cont.
PCI-ISA
Bridge
PIC,
PIT,
DMA
VRAM
Graphics
LCD
CRT
Primary PCI BUS
ISA BUS
BM-IDE
cont.
Primary
channel
HDD
0
Secondary channel
HDD
CD-ROM
Swappable Bay
Docking Station
Docking
PCI/PCI
Bridge
Secondary PCI BUS
PCI Slots

~~1.1.1.2~~6.3.1.2 Filling in the ISA Scope with Static Device Objects

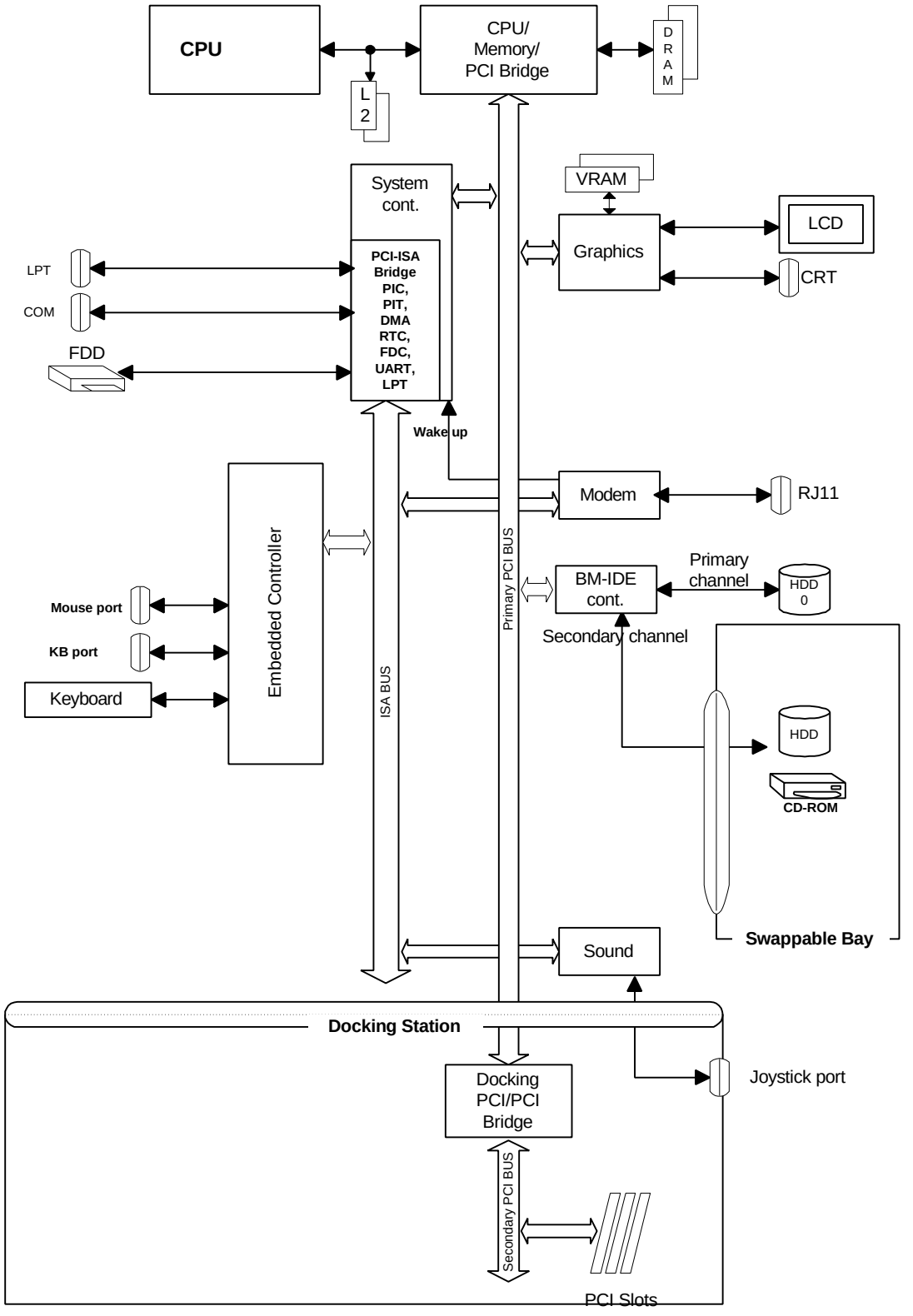
The following ACPI name space model shows the bare bones bus structure with the PC root bus objects that must be in ACPI name space, and then with the ISA bus objects added (the bare bones and PCI root objects are bolded). The ISA bus objects are:

- Embedded controller (object named “EC0” in this example).
- FDD controller (object named “FDC0” in this example).
- Dual-purpose printer port (objects named “LPT” and “ECP” in this example).
- Com port (object named “COM0” in this example).
- Modem (object named “MDM0” in this example).
- Sound device (object named “SND0” in this example).
- Joystick game port (object named “SND1” in this example).

```
PCI0                    //PCI root bridge (Host PCI bridge)
    _HID                //PCI Bus ID (used on root bus only)
    _ADR                //address of PCI device
    _CRS                //report PCI bus number zero (used on root bus only)
    IDE                 //BM-IDE Device(PCI)
        _ADR            //address of PCI device
        PRIM            //Primary Bus Master controller
            _ADR        //Primary channel
        BAY             //swappable bay for 2nd HDD and CD-ROM
            _ADR        //Secondary channel
            _LCK        //means ejectable
  VID0                  //VIDEO Device (PCI)
        _ADR            //address of PCI device
    ISA                 //PCI-ISA Bridge
        _ADR            //Device address of PCI-ISA bridge on PCI bus
        EC0             //Embedded controller device
            _HID        //ID for Embedded controller
        FDC0            //Floppy Disk controller
            _HID        //Hardware Device ID
        LPT             //Standard Printer port
            _HID        //Hardware Device ID
        ECP             //ECP Printer
            _HID        //Hardware Device ID
        COMA            //Communication Device
            _HID        //Hardware Device ID
        MDM0            //Communication Device (Modem)
            _HID        //Hardware Device ID
        SND0            //Sound Device
            _HID        //Hardware Device ID
        SND1            //Joystick (Game port)
            _HID        //Hardware Device ID
            _EJD        //dock dependent device
    DOCK                //PCI Bus 1 of DOCKING STATION (PCI-PCI bridge)

        _ADR            //Device address of the PCI bus
        _UID            //Docking station Unique ID
```

The following version of the mobile concept machine block diagram shows all the physical devices that are represented in this ACPI name space above.


CPU
CPU/
Memory/
PCI Bridge
DRAM
L2
System
cont.
VRAM
Graphics
LCD
CRT
LPT
COM
FDD
PCI-ISA
Bridge
PIC,
PIT,
DMA
RTC,
FDC,
UART,
LPT
Wake up
Modem
RJ11
Embedded Controller
Primary PCI BUS
ISA BUS
BM-IDE
cont.
Primary
channel
HDD
0
Secondary channel
Mouse port
KB port
Keyboard
HDD
CD-ROM
Swappable Bay
Sound
Docking Station
Joystick port
Docking
PCI/PCI
Bridge
Secondary PCI BUS
PCI Slots

~~1.1.1.3~~6.3.1.3 Filling in the _SB Scope with Static Device Objects

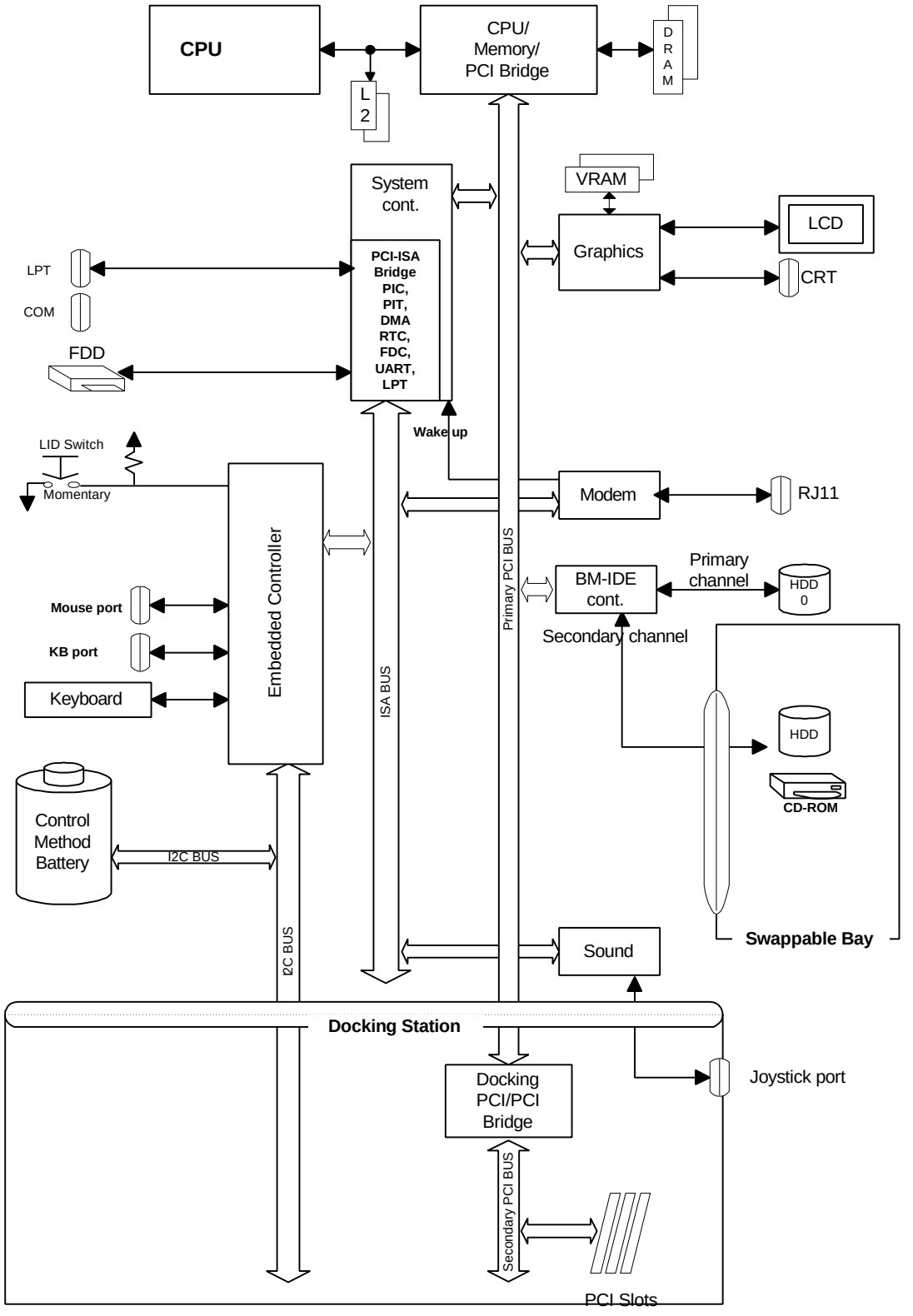
The following ACPI name space model shows the bare bones model with the system bus (_SB) objects added. The system bus objects are:

- Lid switch (named "LID" in this example)
- AC adapter (named "AC" in this example)
- Battery (named "BAT0" in this example)

```
\_SB
    LID                     //LID
        _HID                //LID ID
    BAT0                    //Battery
        _HID                //Battery Device ID
    AC                      //AC Adapter
    PCI0                    //PCI root bridge (Host PCI bridge)
        _HID                //PCI Bus ID (used on root bus only)
        _ADR                //address of PCI device
        _CRS                //report PCI bus number zero (used on root bus only)
        IDE                 //BM-IDE Device(PCI)
            _ADR            //address of PCI device
            PRIM            //Primary Bus Master controller
                _ADR        //Primary channel
            BAY             //swappable bay for 2nd HDD and CD-ROM
                _ADR        //Secondary channel
                _LCK        //means ejectable
      VID0                  //VIDEO Device (PCI)
            _ADR            //address of PCI device
        ISA                 //PCI-ISA Bridge
            _ADR            //Device address of PCI-ISA bridge on PCI bus
            EC0             //Embedded controller device
                _HID        //ID for Embedded controller
            FDC0            //Floppy Disk controller
                _HID        //Hardware Device ID
            LPT             //Standard Printer port
                _HID        //Hardware Device ID
            ECP             //ECP Printer
                _HID        //Hardware Device ID
            COMA            //Communication Device
                _HID        //Hardware Device ID
            MDM0            //Communication Device (Modem)
                _HID        //Hardware Device ID
            SND0            //Sound Device
                _HID        //Hardware Device ID
            SND1            //Joystick (Game port)
                _HID        //Hardware Device ID
                _EJD        //dock dependent device
        DOCK                //PCI Bus 1 of DOCKING STATION (PCI-PCI bridge)

            _ADR            //Device address of the PCI bus
            _UID            //Docking station Unique ID
```

This name space corresponds to the following parts of the mobile concept machine block diagram.


CPU
CPU/
Memory/
PCI Bridge
DRAM
L2
System
cont.
VRAM
Graphics
LCD
CRT
LPT
COM
FDD
PCI-ISA
Bridge
PIC,
PIT,
DMA
RTC,
FDC,
UART,
LPT
Wake up
LID Switch
Momentary
Embedded Controller
Modem
RJ11
Primary PCI BUS
BM-IDE
cont.
Primary
channel
HDD
0
Secondary channel
Mouse port
KB port
Keyboard
ISA BUS
HDD
CD-ROM
Control
Method
Battery
I2C BUS
I2C BUS
Swappable Bay
Sound
Docking Station
Joystick port
Docking
PCI/PCI
Bridge
Secondary PCI BUS
PCI Slots

~~1.4~~6.4 Adding Dynamic Event Handling to the ACPI Name Space

The name space model of the mobile concept machine that exists after including all the device objects and device identifier objects is a static set of objects. This section describes how objects are used in ACPI name space to handle dynamic events.

The following dynamic events can take place on the mobile concept machine platform:

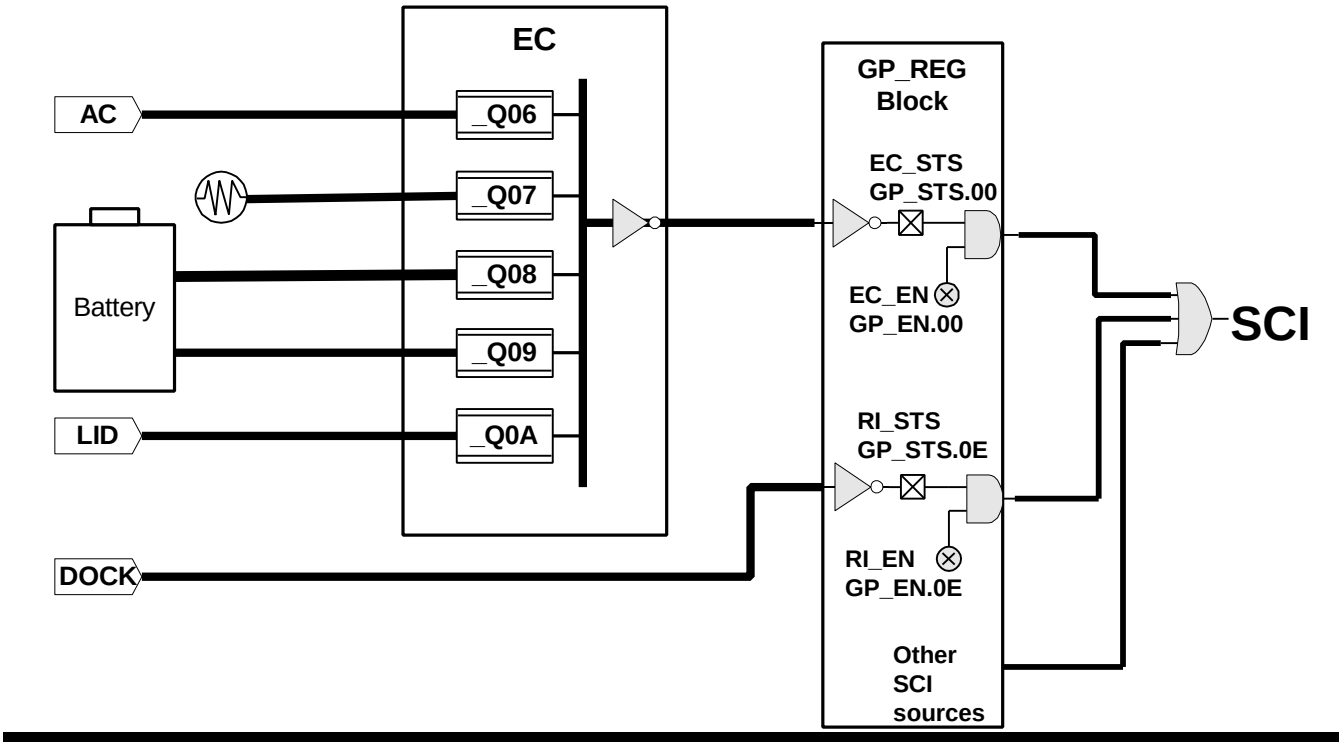
- The embedded controller can detect:
- A lid switch event (mobile platform lid opens or closes).
- An AC adapter event (AC adapter is plugged in or unplugged).
- A battery event (battery low warning, battery critical warning, etc.).
- A thermistor event (high temperature warning, etc.).
- A swappable device can be inserted into or ejected from the bay.
- The mobile platform can be docked or undocked.
- The modem can wakeup the system.
- Video power saving mode can come on or go off.
- The user can press the power/power override button.

In addition to these specific dynamic events, other changes can take place on the platform:

- The status of various devices can change (for example, the joystick (SND1) or the LPT/ECP port).
- The resource settings of various devices can change (for example, the Com port address or the SND0 DMA).

6.4.1 Use of an Embedded Controller on the Mobile Concept Machine

The following block diagram shows the relationships between the devices that are wired to the embedded controller, the embedded controller queries, and the ACPI-specified General Purpose Register block which is the source of an SCI. All embedded controller events raise the GP_STS.00 bit in the GP_REG block, and if that bit is enabled, and SCI occurs. The control method that handles an SCI from GP_STS.00 will notify the EC0 device, and the OS will query the EC to determine whether to run the _Q06, _Q07, _Q08, or _Q0A control method depending upon whether the embedded controller event is from the AC adapter, thermistor, battery, or lid, respectively.



Note that in the preceding block diagram, a non-embedded controller event, the docking event, is shown tied to GP_STS.0E; this is a general-purpose event. AC adapter, thermistor, battery, and lid events could all be handled as general-purpose events, also. Using an embedded controller on a platform is totally optional. Some advantages of using an embedded controller are:

- More flexible design.
- Enables use of I2C bus communication.

6.4.1.1 Embedded Controller Operation Region and Fields

The following ASL code defines the operation region and fields that dynamically track events on the mobile concept platform. The fields that are used to handle lid and docking events are shown in bold typeface.

```
//create EC's region and field
OperationRegion(RAM, EmbeddedControl, 0, 0xFF)
    Field(RAM, AnyAcc, Lock, Preserve) {
        // Fields for System Indicators
        NMSG,   8,    // Number of Message appeared on Message indicator
        SLED,   4,    // System Status indicator
                      // bit 3: System is Working
                      // bit 2: System is waking up
                      // bit 1: System is sleeping (S1,S2 or S3)
                      // bit 0: System is sleeping with context saved (S4).
        ,4,           // reserved
        // Fields for FAN information placed here
        MODE,   1,    // thermal policy (quiet/perform)
        FAN,1,  // fan power (on/off)
        TME0,   1,    // require notification with 0x80
        TME1,   1,    // require notification with 0x81
        ,2,           // reserved
        // Fields for Thermal information placed here
        AC0, 16,  // active cooling temp
        PSV, 16,  // passive cooling temp
        CRT,16,  // critical temp
        TMP,    16,   // current temp
        // Fields for LID and LCD information placed here
        LIDS,   1,    // LID status
        LSW0,   1,    // LCD power switch
        // wake up enable, disable
        LWKE,   1,    // Enable wake up from LID
        WAKF,   1,    // Indicates wake has failed
        ,3,           // reserved
        MWKE,   1,    // Enable wake up from MODEM
        ,4,           // reserved
        // sleep type
        SLPT,   8,    // Set sleep type before system enter
                      // the sleep state.  This field will
                      // used in the _PTS control method
        // docking information is placed here
        DCID,   32,   // Docking unique ID
        DSTS,   1,    // Docking status
        UDRS,   1,    // UNDOCK_REQUEST_STS
        DCS,    1,    // DOCK_CHG_STS
        UDW,    1,    // UNDOCK_WARM
        UDH,    1,    // UNDOCK_HOT
        DCCH,   1,    // DOCKING STATUS has been changed
        ,1,           // reserved
        // SWAPPABLE BAY's information is placed here
        SWEJ,   1,    // SWAPPABLE BAY eject request
        SWCH,   1,    // condition of SWAPPABLE BAY was changed
        ,6,           // Reserved
        //
        // AC and CMBatt information is placed here
        //
        ADP,1,  // AC Adapter 1:On-line, 0:Off-line
        AFLT,   1,    // AC Adapter Fault  1:Fault  0:Normal
        BAT0,   1,    // BAT0  1:present, 0:not present
        ,1,           // reserved
        BPU0,   32,   // Power Unit
        BDC0,   32,   // Designed Capacity
        BFC0,   32,   // Last Full Charge Capacity
        BTC0,   32,   // Battery Technology
        BDV0,   32,   // Design Voltage
        BST0,   32,   // Battery State
        BPR0,   32,   // Battery Present Rate
                      // (Designed Capacity)x(%)/{(h)x100}
        BRC0,   32,   // Battery Remaining Capacity
                      // (Designed Capacity),˜(%)•  ^100
        BPV0,   32,   // Battery Present Voltage
        BTP0,   32,   // Trip Point
        BCW0,   32,   // Design capacity of Warning
        BCL0,   32,   // Design capacity of Low
        BCG0,   32,   // capacity granularity 1
        BG20,   32,   // capacity granularity 2
```

```
        BMO0,     32,   // Battery model number field
        BIF0,     32,   // OEM Information(00h)
        BSN0,     32,   // Battery Serial Number
        BTY0,     64    // Battery Type (e.g., "Li-Ion")
} // end field
```

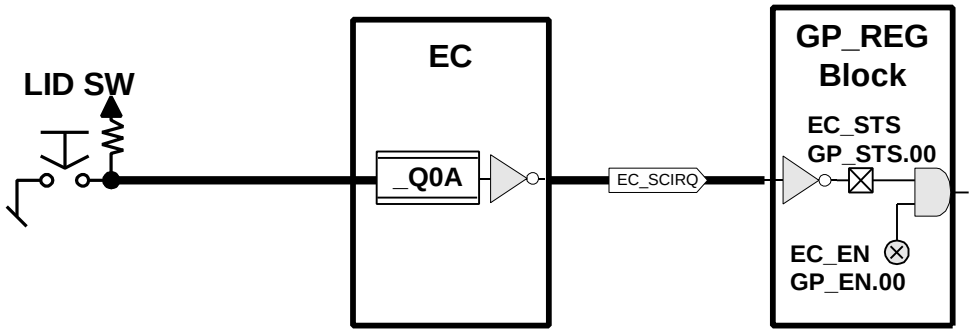
~~1.1.1.2~~6.4.1.2 Handling Embedded Controller Lid Switch Events

A lid switch event happens when the mobile platform lid opens or closes.

The following objects in ACPI name space are involved in handling lid events as embedded controller events:

```
\_GPE                           //General Purpose event
    _L00                    //EC control Method to handle GP_STS.00
    .
    .
    .
\_SB
    LID                         //LID
        _PRW                    //define wake device
        _LID                //status of LID
        _PSW                    //enable/disable lid wake
    .
    .
    .
    PCI0                    //PCI root bridge (Host PCI bridge)
        .
        .
        .
        ISA                     //PCI-ISA Bridge
            EC0                 //Embedded controller device
                .
                .
                .
                _Q09            //LID event notification
                .
                .
                .
```

The lid switch and its relationship to the embedded controller and GP_REG block are shown in the following diagram:



The sequence of steps in handling a lid closing event are listed below:

1. The mobile platform user closes the lid.
2. An SCI fires (see block diagram above).
3. The OS's policy for device checking an embedded controller is to query the device using the standard embedded controller query interface. Since this sequence started with the user closing the lid, the embedded controller returns the value of 0A in response to the query.
4. An _GPE is needed under the EC device indicating which bit index the EC is connected to.

5. This causes the OS to run the _Q0A event handler. ASL code for the _Q0A event handler is shown below.

```
// Lid event - EC query value A
Method(_Q0A) {
      Notify (\_SB.LID, 0x80)            // notify LID status changed
}
```

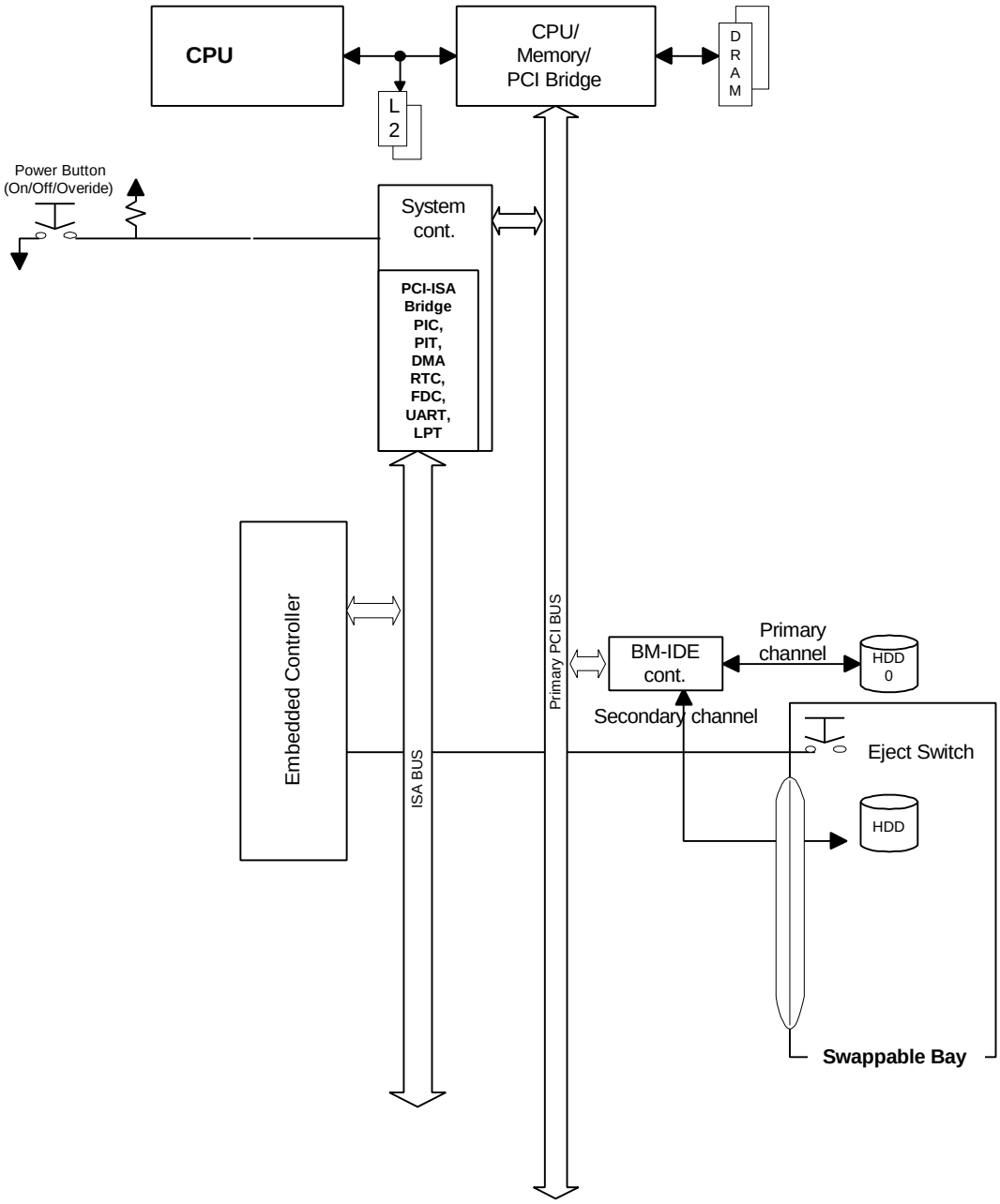
6. The _Q0A event handler sends a "lid status change" notification to the OS, naming the device object _SB.LID.
7. The OS responds to this notification by running the _LID control method, which is defined in the ACPI Specification, Revision 1.0, to always return the status of the lid. The ASL code for the _LID method for the mobile concept machine is:

```
Method(_LID) {
      Return( \_SB.PCI0.ISA.EC0.LIDS)      // Status of the LID
            }
```

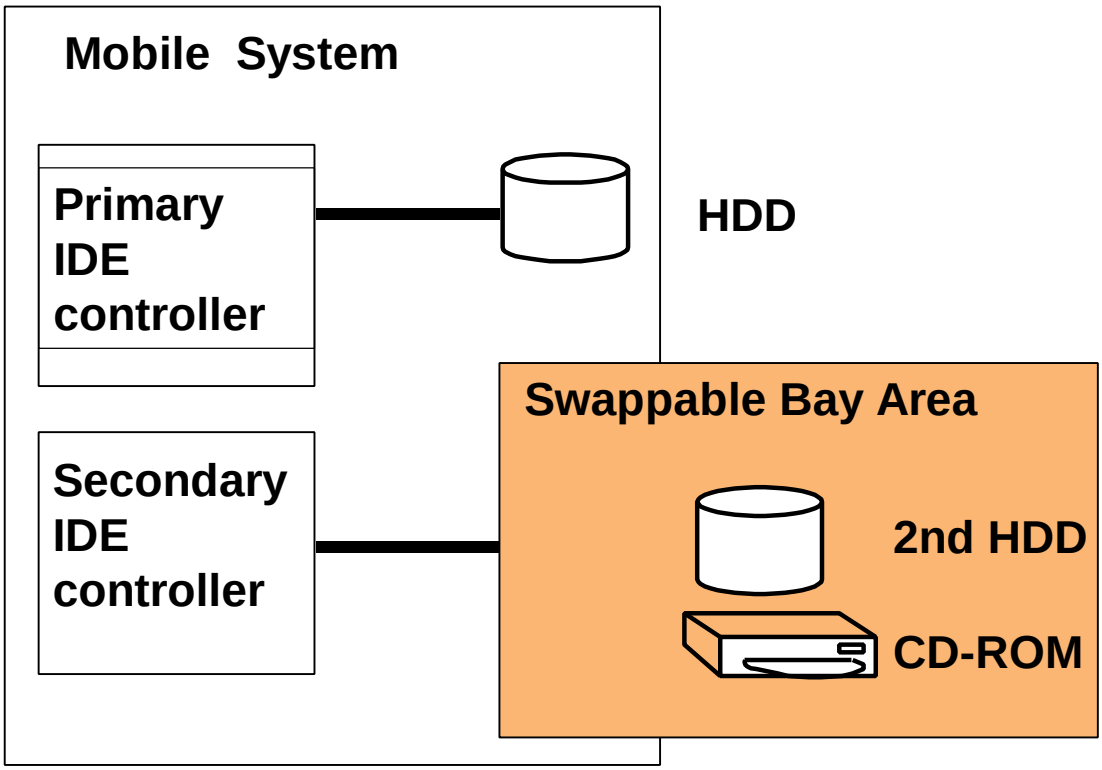
7. "LIDS" is the name of a field in the embedded controller operation region which always contains the status of the lid. In this scenario, since the lid is closed, the value returned to the OS by the _LID method is zero.
8. The OS carries out its "closed lid" policy.

6.4.2 Handling Device Swapping in the Bay

A swappable device can be inserted into or ejected from the bay. The mobile concept machine bay and its relationship to the other mobile platform components is shown in the following hardware block diagram.



This can be shown more simply in the following logical block diagram.



When a second HDD device is inserted in the Bay, two IDE controllers and two IDE HDD devices are active. Which IDE controller controls the primary channel and which IDE controller controls the secondary channel is encoded in the _ADR object under each IDE controller's Device object in the ACPI name space, using the following convention:

```
Name(_ADR, 0)     //Primary channel
Name(_ADR, 1)     //Secondary channel
Name(_ADR, 2)     //Third channel (if needed)
Name(_ADR, 3)     //Fourth channel (if needed)
```

Which IDE device is the master and which is the slave is encoded in the _ADR object under each IDE HDD's Device object in the ACPI name space, using the following convention:

```
Name(_ADR, 0)     //Master
Name(_ADR, 1)     //Slave
```

These conventions are used in the following block of code from the mobile concept machine's ASL code:

```
IDE                     //BusMaster IDE controller
    _ADR                //PCI address of BM-IDE
    PRIM                //Primary Bus Master controller
        _ADR            //Primary channel
    BAY                 //swappable bay for 2nd HDD and CD-ROM
        _ADR            //Secondary channel
        _LCK            //means ejectable
```

The following ASL code implements the namespace shown above.

```
        //*************************************************
// BusMaster IDE
//*************************************************
Device(IDE) {                                   //BM-IDE in system
  Name(_ADR, 0x00010000)                        //PCI address of BM-IDE (dev, func)
        Method(_STA,0) {                        //Status of BM-IDE controller
    // If BM-IDE is functioning
    Return(0xF)
    // If IDE channel0 is disabled
    Return(0xD)
  }

        Device(PRIM) {
            Name(_ADR,0)                        //Primary IDE channel
    Method(_STA,0) {                            //Status of the primary channel
      // If IDE is exist and functioning
      Return(0xF)
      // If IDE channel0 is removed
      Return(0xD)
    }
  } // end PRIM

  Device(BAY) {                                 //secondary IDE for BAY
    Name(_ADR, 1)                               //secondary IDE channel
            Method(_STA,0) {                    //Status of secondary channel
      // If IDE is exist and functioning
      Return(0xF)
      // If IDE channel1 is removed
      Return(0xD)
    }
    Method(_LCK,1){                             //means ejectable
      // Lock or unlock the SWAPPABLE BAY
      If (ARG0) {
        Store (0x1, \_SB.PCI0.ISA.EC0.SWEJ) //lock
      } Else {
        Store (0x0, \_SB.PCI0.ISA.EC0.SWEJ) //unlock
      }
    }
        } // end BAY
} // end IDE
```

Following is the Bay event handler code.

```
        // BAY changed event - EC query value B
        Method(_Q0B) {
            //When SWAPPABLE BAY is attached or unattached
            //this event will happen.
            Notify(\_SB.PCI0.ISA.BAY, 0)    // bay event
        }
        // BAY eject request event - EC query value C
        Method(_Q0C) {
            //When eject switch for SWAPPABLE BAY is pressed
            //this event will happen.
            Notify(\_SB.PCI0.ISA.BAY, 1) // bay eject request
}
```

~~1.1.3~~6.4.3 Interrupt Sharing

On the mobile concept machine, this would provide an extra PCI interrupt once Windows 98 has started, useful for Cardbus insertions and similar operations. Note that:

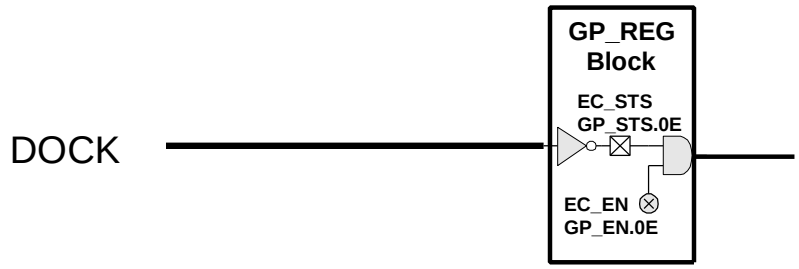
- The ACPI devnode for the math coprocessor (PNP0C04) needs to declare the interrupt as shareable AND level trigger (the attribute byte needs to be 0x18, just like the PIRQ devices.)
- IRQ 13 can be in the PRS of the PIRQ device (or any other device that supports shareable level trigger mode), but not in the CRS. Since real mode may still be running, VMCPD cannot be allowed to end the chain (it always consumes the chain and therefore never gets reflected to real mode.) In other words, IRQ 13 can only be used by MCP and for ACPI interrupt, until the operating system decides to rebalance.

6.4.4 Handling Dock Events

A dock event occurs when the user docks or undocks the mobile platform, or when the user makes an undocking request.

~~1.1.1.1~~6.4.4.1 Handling Dock Events as General Purpose Events (GPEs)

On the mobile concept machine, docking events are handled as a General Purpose Event (GPE). Docking events are tied to a bit in the GP_REG block as shown below.



~~1.1.1.2~~6.4.4.2 ACPI Name Space Objects that Handle Dock Events

The following objects in ACPI name space are involved in handling dock events as embedded controller events:

```
\_GPE                           //General Purpose event
    .
    .
    .
    _L0E                        //docking method to handle GP_STS.0E
    .
    .
    .
\_SB
    LNKA                        //PCI Interrupt routing
        _HID
        _UID
        _PRS
        _DIS
        _CRS
        _SRS
    LNKB                        //PCI Interrupt routing
        _HID
        _UID
        _PRS
        _DIS
        _CRS
        _SRS
    LNKC                        //PCI Interrupt routing
        _HID
        _UID
        _PRS
        _DIS
        _CRS
        _SRS
    LNKD                        //PCI Interrupt routing
        _HID
        _UID
        _PRS
        _DIS
        _CRS
        _SRS
    .
    .
    .
    PCI0                    //PCI root bridge (Host PCI bridge)
        .
        .
        .
        ISA                 //PCI-ISA Bridge
            EC0             //Embedded controller device
                .
                .
                .
                _Q0A        //Docking event notification
                .
                .
                .
            SND0            //Sound Device
                _HID        //Hardware Device ID
                _STA        //Status of the Communication Device
                _DIS    //Disable
                _CRS        //Current Resource
                _PRS        //Possible Resource
                _SRS        //Set Resource
            SND1            //Joystick (Game port)
                _HID        //Hardware Device ID
                _EJD        //dock dependent device
                _STA        //Status of the Communication Device
                _DIS    //Disable
                _CRS        //Current Resource
                _PRS        //Possible Resource
                _SRS        //Set Resource
        DOCK                //PCI Bus 1 of DOCKING STATION (PCI-PCI bridge)
                            //Fully qualified name is \_SB.PCI0.DOCK
```

```
_ADR            //Device address of the PCI bus
_UID            //Docking station Unique ID
_STA            //Status of the Docking Station
_EJ0        //Eject with S0 state
_EJ3        //Eject with S3 state
_PRT            //PCI IRQ routing information
```

The object named "DOCK" in the name space above is an ACPI Device object because the dock device is just a bus bridge (typically, PCI bus #1 or greater). The block of ASL code that declares the Device object named "DOCK" for the mobile concept machine is shown below.

```
        //*******************************
// Dock Objects (PCI-PCI Bridge)
//*******************************
Device(DOCK) {                                  // PCI-PCI bridge
   Name (_HID, EISAID("PNP0A03"))               //_HID for PCI bus
   Name (_ADR,0x0004FFFF)                       // PCI device#,func#
   Method (_UID) {                              //Docking station unique ID
      Return (\_SB.PCI0.ISA.EC0.DCID)
   }
   Method (_STA) {              //Status of the DOCK
      if (\_SB.PCI0.ISA.EC0.DSTS) {     //if docked
         Return(0x0F)           //return functioning
      } else {              //if undocked
         Return(0x00)           //return not exist
      }
   }
   Method (_EJ0, 1) {                           //supports S0 (hot) undock
      //Store(0x0, \_SB.PCI0.ISA.EC0.UDR)
      Sleep(1000)
   }
   Method (_EJ3, 1) {                           //supports S3 undock
      Store (0x01, \_SB.PCI0.ISA.EC0.UDW)
   }
   // PCI SLOT IRQ routing
   Name(_PRT, Package(){
      Package(){0x0001ffff, 0, LNKA, 0},  // Slot 1, INTA
      Package(){0x0001ffff, 1, LNKB, 0},  // Slot 1, INTB
      Package(){0x0001ffff, 2, LNKC, 0},  // Slot 1, INTC
      Package(){0x0001ffff, 3, LNKD, 0},  // Slot 1, INTD
      }
   ) //end _PRT
} // end DOCK
```

~~1.1.5~~6.4.5 Walking Through a Dock Event

The sequence of steps in handling a docking event are listed below:

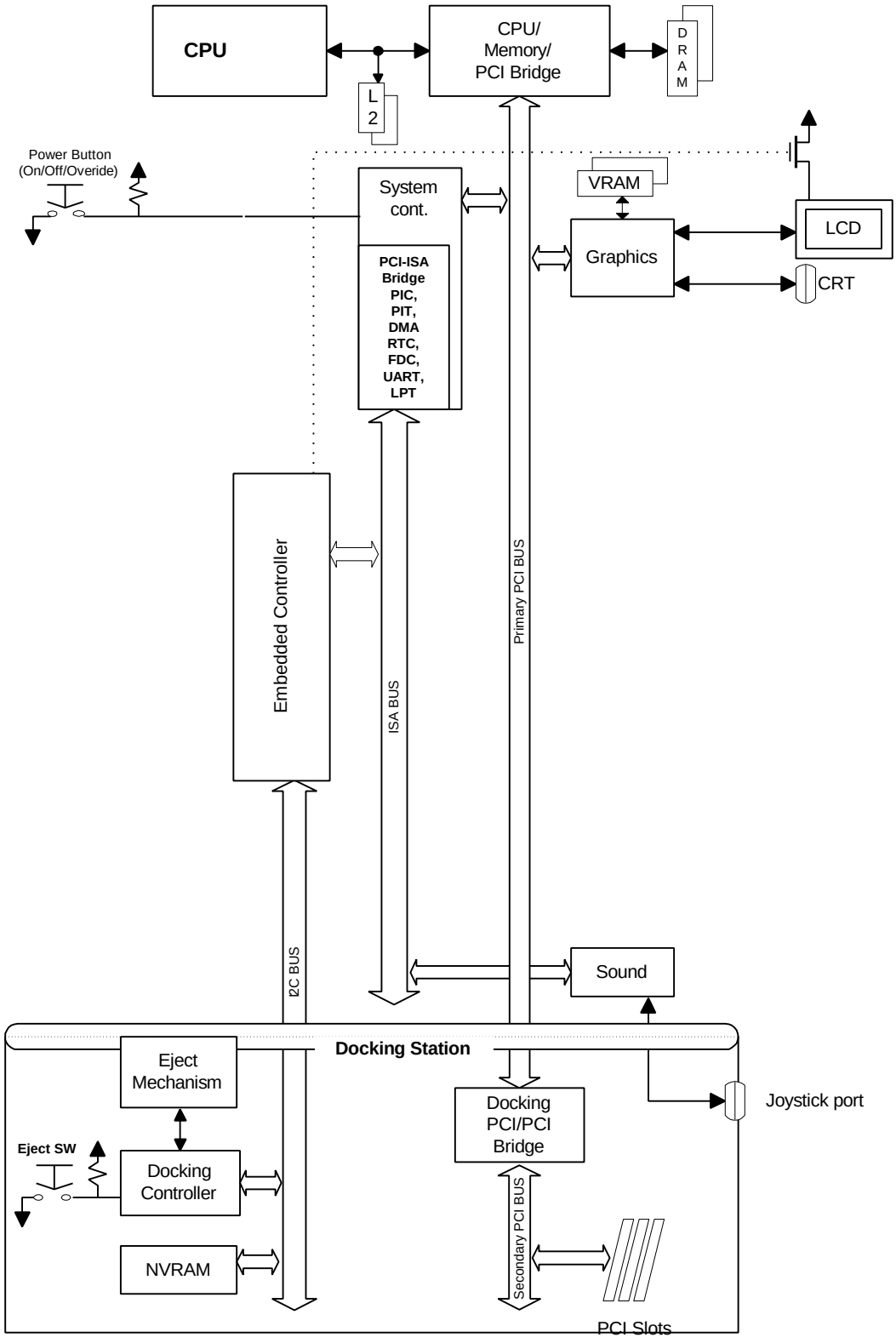
1. The mobile platform user plugs the mobile platform into the dock (for example).
2. An SCI fires (see block diagram above).
3. The OS detects GPE 0E and runs the _L0E event handler. ASL code for the _L0E event handler for the mobile concept machine is shown below (line numbers added). Notice the use of fully qualified name space path names in the code below and compare these names to the name space map in the previous section; use fully qualified names the first time you write a block of code to avoid name scope errors.

```
Method (_L0E) {                                 // GP event handle to GP_STS.0E
        IF (\_SB.PCI0.ISA.EC0.UDW) {
           Notify (\_SB.PCI0.DOCK, 0)           // Warm Undocked
           Store (0, \_SB.PCI0.ISA.EC0.UDW)
        }
        IF (\_SB.PCI0.ISA.EC0.UDH) {
           Notify (\_SB.PCI0.DOCK, 0)           // Hot Undocked
           Store (0, \_SB.PCI0.ISA.EC0.UDH)
        }
        IF (\_SB.PCI0.ISA.EC0.UDRS) {
           Notify (\_SB.PCI0.DOCK, 1)           // about to Hot undock
           Store (0, \_SB.PCI0.ISA.EC0.UDRS)
        }
        IF (\_SB.PCI0.ISA.EC0.DCS) {
           Notify (\_SB.PCI0.DOCK, 0)           // Docked
           Store (0, \_SB.PCI0.ISA.EC0.DCS)
        }
     } // end _L0E
```

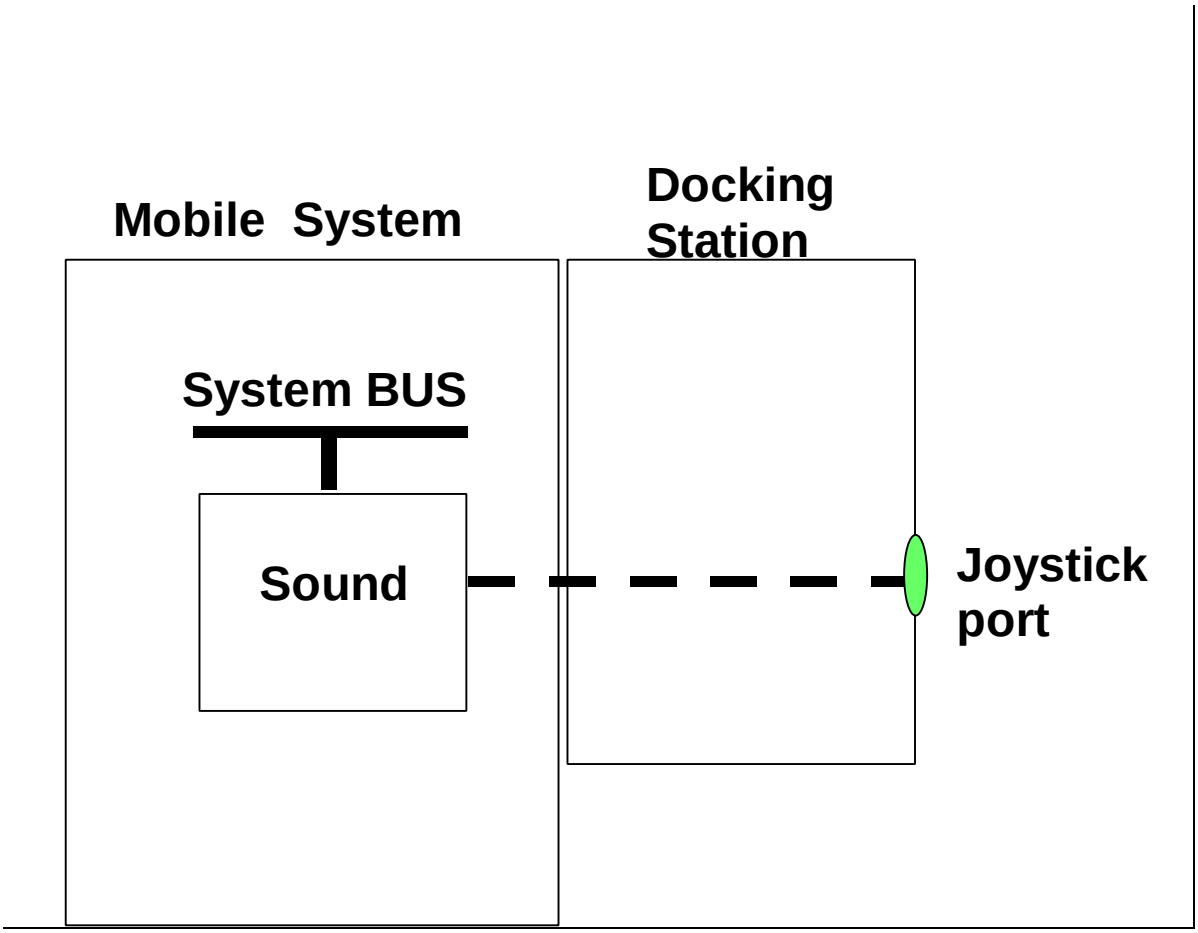
4. In this example, where the user has docked the mobile computer, lines 14 through 17 in the _L0E method are executed and the _L0E event handler sends a “device check” notification to the OS, naming the device to check (the fully-qualified name of the dock in the hierarchical ACPI name space is _SB.PCI0.DOCK).
5. The OS runs its docking policy.

6.4.6 Device Status Changes

The status of various devices can change. The most obvious example of this on the mobile concept machine is the joystick device. A joystick is only available to the Mobile concept machine user when the mobile platform is docked, as shown in the following hardware block diagram.



This relationship can be shown more simply with the following logical block diagram.



~~1.1.7~~6.4.7 ACPI Name Space for the Joystick Device

The following objects in ACPI name space are managing the changing status of the joystick device as it comes and goes with the dock.

In the ACPI namespace that follows, the joystick device is represented by the Device object named ‘SND1’. The status of the joystick device (that is, whether the joystick device is functioning for the mobile machine user as the user docks and undocks the mobile machine) is reported by the _STA object under the SND1 Device object in the hierarchical ACPI name space.

```
\_GPE                           //General Purpose event
    _L0E                        //docking method to handle GP_STS.0E
    .
    .
    .
\_SB
    .
    .
    .
    PCI0                    //PCI root bridge (Host PCI bridge)
        .
        .
        .
        ISA                     //PCI-ISA Bridge
            _HID                //PNPID for ISA bus
            EC0                 //Embedded controller device
                .
                .
                .
                _Q0A            //Docking event notification
                .
                .
                .
            SND0                //Sound Device
                _HID            //Hardware Device ID
                _STA            //Status of the Communication Device
                _DIS        //Disable
                _CRS            //Current Resource
                _PRS            //Possible Resource
                _SRS            //Set Resource
            SND1                //Joystick (Game port)
                _HID            //Hardware Device ID
                _EJD            //dock dependent device
                _STA            //Status of the Communication Device
                _DIS        //Disable
                _CRS            //Current Resource
                _PRS            //Possible Resource
                _SRS            //Set Resource
        DOCK                    //PCI Bus 1 of DOCKING STATION (PCI-PCI bridge)
            _HID                //PNPID for PCI bus
            _ADR                //Device address of the PCI bus
            _UID                //Docking station Unique ID
            _STA                //Status of the Docking Station
            _EJ0            //Eject with S0 state
            _EJ3            //Eject with S3 state
            _PRT                //PCI IRQ routing information
```

~~1.1.8~~6.4.8 Sample ASL Code for the Joystick Device

Following is the sample ASL code for the SND0 and SND1 Device objects. Notice the code in the _STA method for SND1, which controls whether or not the joystick device appears in the mobile machine UI as the mobile machine docks and undocks.

```
                    //
// Sound Devices
//
Device(SND0) {                      // Sound Device
                                    // (WSS+FM+etc)
    Name(_HID,EISAID("SND0000"))    // example ID for Sound
    Method(_STA,0) {                // Status of the sound
      //When functioning
      Return (0xF)                  // device is functioning
      //When disabled by OS,
      Return (0xD)
    }
    Method (_CRS) {                 // Current Resources
      //Prepare the current resource of UART
      //Name (BUFF, buffer(size){data})
      //Return (BUFF)
    }
    Method (_PRS) {                 // Possible Resources
      //Prepare the current resource of UART PORT
      //Name (BUFF, buffer(size){data})
      //Return (BUFF)
    }
    Method (_SRS,1 ) {              // Set Resources
      // ARG0 = PnP Resource String to Set
      //Control of setting resource is placed here
    }
    Method (_DIS,0 ) {              // Disable Resources
      //Control of setting resource is placed here
    }
} // end SND0
//
// Joystick port is on the docking station.
// Joystick will only appear in UI when docked.
//
Device(SND1) {                                  // Joystick (Game Port)
    Name(_HID,EISAID("SND0001"))                // example ID for Joystick
    Name(_EJD,"\_SB.PCI0.DOCK")                 // means dock-dependent device
    Method(_STA,0) {                            // Status of the Joystick
      if (\_SB.PCI0.ISA.EC0.DSTS) {     // docked and functioning
        Return (0x0F)                           // return show UI
      } else {                                  //When undocked and not shown in UI
        Return (0x0B)                           //return remove UI
      }
    }
    Method (_CRS) {                             // Current Resources
      //Prepare the current resource of UART
      //Name (BUFF, buffer(size){data})
      //Return (BUFF)
    }
    Method (_PRS) {                             // Possible Resources
      //Prepare the current resource of UART PORT
      //Name (BUFF, buffer(size){data})
      //Return (BUFF)
    }
    Method (_SRS,1 ) {                          // Set Resources
      // ARG0 = PnP Resource String to Set
      //Control of setting resource is placed here
    }
    Method (_DIS,0 ) {                          // Disable Resources
      //Control of setting resource is placed here
    }
} // end SND1
```

~~1.1.9~~6.4.9 Device Resource Setting Changes

The resource settings of various devices can change. This is done by using the _PRS, _SRS, and _CRS objects that are under a Device object that represents a device with more than one set of resource settings.

~~1.5~~6.5 Complete Mobile Concept Machine ACPI Name Space

This section shows the hierarchy of objects in ACPI name space that models the block diagram shown above.

```
\_PR
    CPU0
\_S0                    //Value for S0 to set the SLP_TYP
\_S2                    //Value for S2 to set the SLP_TYP
\_S3                    //Value for S3 to set the SLP_TYP
\_S5                    //Value for S5 to set the SLP_TYP
\_PTS                       //Prepare to sleep control method
\_WAK                       //Wake
\_SI                    //System Indicator (ICON)
    _MSG                    //Message waiting indicator
    _SST                    //System status indicator
\_TZ                    //Thermal zone
    THRM                    //Thermal zone
        _TMP
        _AC0
        _AL0
        _PSV
        _PSL
        _CRT
        _SCP
        _TC1
        _TC2
        _TSP
    PFAN                    //FAN power resource
        _STA
        _ON
        _OFF
    FAN                     //FAN Device
        _HID                //
        _PR0                //list of power resource
\_GPE                       //General Purpose event
    _L00                //EC control Method to handle GP_STS.00
    _L0E                    //docking method to handle GP_STS.0E
    _L0F                //Ring wake method to handle GP_STS.0F
\_SB
    LNKA                    //PCI Interrupt routing
        _HID
        _UID
        _PRS
        _DIS
        _CRS
        _SRS
    LNKB                    //PCI Interrupt routing
        _HID
        _UID
        _PRS
        _DIS
        _CRS
        _SRS
    LNKC                    //PCI Interrupt routing
        _HID
        _UID
        _PRS
        _DIS
        _CRS
        _SRS
    LNKD                    //PCI Interrupt routing
        _HID
        _UID
        _PRS
        _DIS
        _CRS
        _SRS
    LID                     //LID
        _HID                //LID ID
        _PRW                //define wake device
        _LID            //status of lid
        _PSW                //enable/disable lid wake
    BAT0                    //Battery
        _HID                //Battery Device ID
        _PCL                //Points to DEVS
```

```
        _STA                    //Status of the battery
        _BIF                //Battery static Information
        _BST                    //Battery present Status
        _BTP                    //Battery Trip point
AC                              //AC Adapter
        _STA                    //AC status
        _PSR                    //Power Source type
        _PCL                    //Power Class List (points to BAT0)

PCI0                        //PCI root bridge (Host PCI bridge)
        _HID                    //PCI Bus ID (Used on the root bus only)
        _ADR                    //address of PCI device
        _CRS                    //report PCI bus number zero (used on the root bus only)
        IDE                     //BM-IDE controller
            _ADR                //PCI address of BM-IDE controller
            _STA                //Status of BM-IDE controller
            PRIM                //Primary Bus Master controller
                _ADR            //Primary channel
                _STA            //Status of the IDE controller
            BAY                 //swappable bay for 2nd HDD and CD-ROM
                _ADR            //Secondary channel
                _STA            //status of the IDE
                _LCK            //Locking control for ejectable device
    VID0                    //VIDEO Device (PCI)
            _ADR                //address of PCI device
            _STA                //Status of VIDEO controller
            _PR0                //Power Management object
            VS0                 //Power Resource
                _STA            //Status of power resource
                _ON             //Power resource on  method
                _OFF            //Power resource off method
        ISA                     //PCI-ISA Bridge
            _ADR                //Device address of PCI-ISA bridge on the PCI bus
            EC0                 //Embedded controller device
                _HID            //ID for Embedded controller
                _STA            //Status of the Device
                _CRS            //Current Resource
                _Q07            //Thermal change notification
                _Q08            //Battery change notification
                _Q09            //Battery status notification
                _Q0A            //LID event notification
                _Q0B            //bay event notification
    _Q0C        //bay eject request notification
            FDC0                //Floppy Disk controller
                _HID            //Hardware Device ID
                _STA            //Status of the Communication Device
                _DIS        //Disable
                _CRS            //Current Resource
                _PRS            //Possible Resource
                _SRS            //Set Resource
            LPT                 //Standard Printer port
                _HID            //Hardware Device ID
                _STA            //Status of the printer device
                _DIS        //Disable
                _CRS            //Current Resource
                _PRS            //Possible Resource
                _SRS            //Set Resource
            ECP                 //ECP Printer
                _HID            //Hardware Device ID
                _STA            //Status of the printer device
                _DIS        //Disable
                _CRS            //Current Resource
                _PRS            //Possible Resource
                _SRS            //Set Resource
            COMA                //Communication Device
                _HID            //Hardware Device ID
                _STA            //Status of the Communication Device
                _DIS        //Disable
                _CRS            //Current Resource
                _PRS            //Possible Resource
                _SRS            //Set Resource
```

```
        MDM0            //Communication Device (Modem)
            _HID        //Hardware Device ID
            _STA        //Status of the Communication Device
            _DIS    //Disable
            _CRS        //Current Resource
            _PRS        //Possible Resource
            _SRS        //Set Resource
            _PR0        //Power resource for D0 state if needed
            _PSW        //Wake-up enable/disable
            _PRW        //Wake-up control method
        SND0            //Sound Device
            _HID        //Hardware Device ID
            _STA        //Status of the Communication Device
            _DIS    //Disable
            _CRS        //Current Resource
            _PRS        //Possible Resource
            _SRS        //Set Resource
        SND1            //Joystick (Game port)
            _HID        //Hardware Device ID
            _EJD        //dock dependent device
            _STA        //Status of the Communication Device
            _DIS    //Disable
            _CRS        //Current Resource
            _PRS        //Possible Resource
            _SRS        //Set Resource
    DOCK                //PCI Bus 1 of DOCKING STATION (PCI-PCI bridge)

        _ADR            //Device address of the PCI bus
        _UID            //Docking station Unique ID
        _STA            //Status of the Docking Station
        _EJ0        //Eject with S0 state
        _EJ3        //Eject with S3 state
        _PRT            //PCI IRQ routing information
```

~~1.6~~6.6 Complete Listing if the Mobile Concept Machine ASL Code

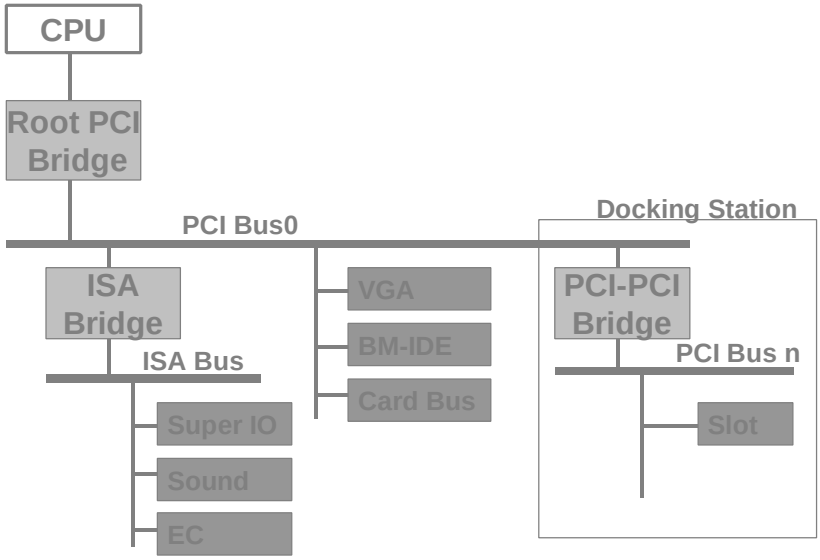
To get a complete listing of the Mobile concept machine sample code, locate the file mb_smp.asl on ACPI Website at http://www.teleport.com/~acpi. Load the file mb_sampl.asl into your programming editor and print out the listing.

The ASL code in the file mb_smp.asl compiles without errors or warnings using version 1.0.3 of the ASL compiler (asl.exe) distributed by Microsoft.

7. A Complete Overview of Docking

Docking functions generally the same way as device insertion/removal. Docking triggers an SCI, which causes a Lxx to be run, which notifies the operating system of the event. The operating system locates the devices, loads the drivers, and completes the docking process. The process is similar for undocking.

This chapter is based on an example system – a mobile computer with a PCI-PCI docking bridge, illustrated in the following figure.



Block Diagram for Mobile Computer Example Using PCI-PCI Docking Bridge

~~1.1~~7.1 Describing Docks in ACPI

Systems that support docking describe their docking capabilities through flags in the FACP and through objects in the ACPI name space. Any machine that supports having a docking station must set the dock capability flag (DCK_CAP) in the FACP. This flag indicates that the operating system should be prepared for docking events.

Once this bit is set, a device must be chosen to represent the docking station in ASL. For this particular design, using a PCI bridge, it is logical to choose the docking bridge as a dock device. For designs that do not use a docking bridge, any one of the devices in the dock can be arbitrarily chosen to represent the dock. Under the dock device that you choose, the following must be listed: the EJx methods, the _STA methods, and the

_DCK and _BDN methods (Note: these two are new methods and will be discussed later in this chapter). Again,

- Set DCK_CAP in the FACP
- Designate the docking bridge as dock device
- Include under that device:

    _EJx methods

    _STA

    _DCK

    _BDNRequired only if your system also supports PNPBIOS.

~~1.1.1~~7.1.1 DCK_CAP

DCK_CAP must be set to '1' if the machine has a docking connector on it. Even if the dock is not connected, always set the flag to '1.' Any portable computer that can be plugged into a docking station, should have DCK_CAP hardwired to '1.' This tells the operating system that the machine is capable of docking so we know it *can* dock even though the AML that describes the docking station might not be loaded in the memory yet. An example would be if you have never docked before.

~~1.1.2~~7.1.2 _DCK

_DCK, the dock control method, has two purposes. First it indicates which device is the docking station and when to display the docking station user interface. Secondly, it controls the bus that goes through the dock connector. When you first plug in your machine, the AML needs to notify the operating system that the machine is docked, but it should not yet have the bus going through the docking connector enabled. This allows the operating system to prepare for devices that might lie on the other side of the docking connector.

When the operating system is ready to dock or undock it will run _DCK, passing it an argument of '1' or '0' telling it to connect the bus or to disconnect the bus, respectively. Remember:

1 = connect the bus

0 = disconnect the bus

Once _DCK has been successfully run, we can perform an enumeration. By doing so, any anomalies can be dealt with. Notice that, although the implementation is different, this behavior is analogous to the ABOUT_TO_CHANGE and CONFIG_CHANGED messages in PNPBIOS. The Notify tells the operating system that docking is going to take place.

At this time, the operating system decides to proceed and performs the dock running _DCK.

~~1.1.3~~7.1.3 _BDN

_BDN is another name space object that is needed if the system also supports the Plug and Play BIOS specification. _BDN indicates the dock station IDreturned by the PNPBIOS, for a given docking station. This allows us to associate a docking station found via PNPBIOS and the exact same docking station via ACPI. Since PNPBIOS and ACPI use two *very* different mechanisms to name docks, a method of correlating the two is needed, hence _BDN.

~~1.1.4~~7.1.4 _EJx Methods

The presence of the _EJx methods not only indicates that the device is capable of being ejected, they report the capabilities of the dock and what states it can eject from. An _EJx is also used to control ejection. The operating system runs these control methods to physically eject the device. Note that:

- _EJ0 – indicates the dock can eject in S0 (hot ejection)
- _EJ1-_EJ3 - indicates the dock can eject in S1-S3 (warm ejection)
- _EJ4-_EJ5 - indicates the dock can eject in S4-S5 (cold ejection)

EJ0 immediately ejects a device. EJ1 through EJ5 set a flag to prepare the system so that when the system does go into that system state (S1–S5), then the hardware activates and ejects the device.

One way to perform eject is to use only one _EJx method. However, two EJx methods will also work. This method is discussed in detail in the following section.

~~1.1.1.1~~7.1.4.1 Using Multiple EJx Methods

If you use two EJx methods to eject a device, one of the control methods must be an EJ0. The way that the algorithm works, regarding which of the two EJx methods the operating system turns to, is based on battery capacity. If the battery has more than 10 percent power capacity remaining, the operating system will run EJ0. If there is less than 10 percent capacity, or even no battery power remaining, the operating system will run the alternate EJx method. Note that EJx is run only after the unloading all of the device drivers for device in the dock.

~~1.1.1.2~~7.1.4.2 Using _EJD

_EJD is another object that may be needed. It is only necessary to use _EJD when you have devices that do not appear in the hierarchy beneath your docking device. An example of this would be if there are two buses that pass through the docking connector. In this case, for those devices that are not underneath the dock, you should include an _ EJD that names the docking device containing the device.

~~1.1.5~~7.1.5 ASL Code

The following ASL code describes a name space for the docking station with a very simple list for the things you need for the docking station that is on the PCI-PCI bridge. Note that under PCI0 we have this PCI device, called DOCK, and it has an _ADR so that we can tell which one it is; it has an _UID, the docking station unique number, then it has an EJ0 so that it does hot docking, it's got a _DCK so that we know it is indeed a docking station, as opposed to some other ejectable device, it's got a _BDN so we can associate it, and it's got a _STA to tell us when it's there and not.

```
Scope(\_SB.PCI0) {
      Device(DOCK) {
            Name(_ADR, …)
            Method(_UID, …)
            Method(_EJ0, 0) {…}
            Method(_EJ4, 0) {…}
            Method(_DCK, 1) {…}
            Name(_BDN, …)
            Method(_STA, 0) {…}
      }
}
```

It is also necessary to know when this machine is being docked or undocked. This is accomplished by using _GPEs. And in your _GPE handing routines, you issue notifies on the docks (more detail later).

For a strong docking example described in ASL code, please refer to the sample ASL for the Mobile Concept Machine (mb_smp.asl) found at the ACPI Webiste at http://www.teleport.com/~acpi/. Also refer to the chapter for the Mobile Concept Machine in this *ImpGuide*.

~~1.2~~7.2 Ejecting a Device

The hardware eject button simply asserts a GPE. It does not do anything else. For example, it does not physically eject the device. This causes a Lxx to get run which issues a Notify on the docking station.

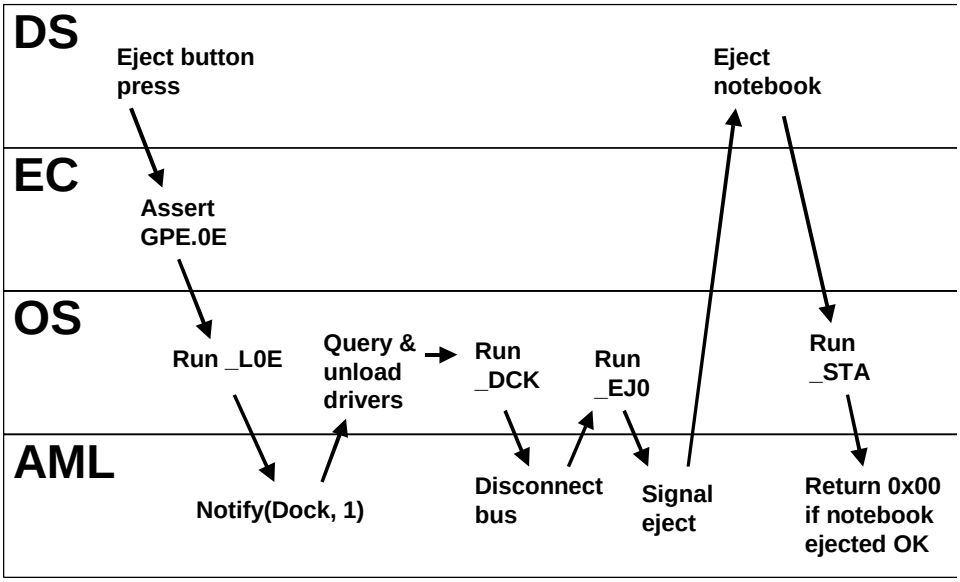
Notify(dock, 1)

This simply says "eject" to the operating system. When the operating system receives this, it instigates the ejection process, unloading device drivers and so on. The operating system also has its own button on the Start menu. That button behaves exactly the same as if the user pushed the hardware button, the difference being that the operating system skips the whole GPE process and just goes straight to the 'undock' code. When you press this button, the operating system performs the ejection on the device object that contains _DCK. This is the other use of _DCK. The following ASL code illustrates the process.

```
Scope(\_GPE) {
      Method(_L0E, 0) {                          // signal dock arrival
            Notify(\_SB.PCI0.DOCK, 0)
}
      Method(_L0F, 0)                            // signal eject button pressed
            Notify(\_SB.PCI0.DOCK, 1)
      }
}
```

~~1.1.1~~7.2.1 Hot Eject Process

The following diagram illustrates what the docking station, the embedded controller, the operating system, and the AML all do over time in the hot eject process. The eject button is pressed, which causes a GPE, which runs a control method, which performs a notify, and the operating system then queries and unloads all the devices found underneath. If the operating system fails to query, then the process stops before proceeding to unload all the device drivers.

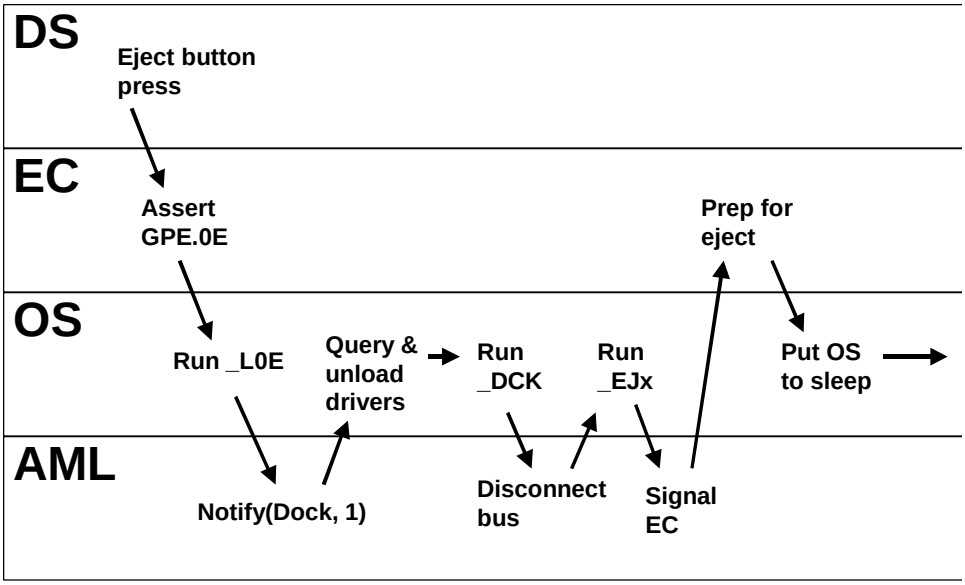


Hot Eject Process

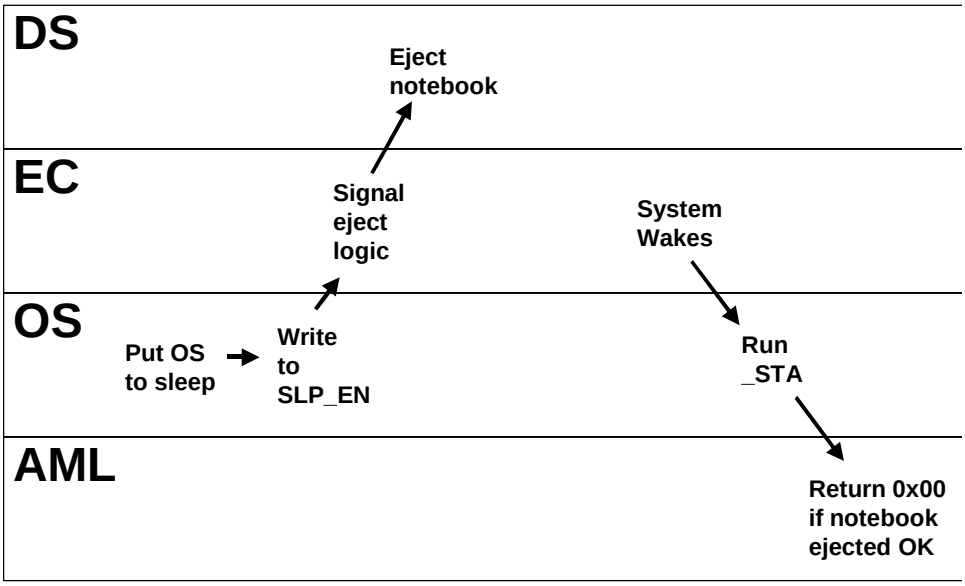
If the operating system decides to continue, it unloads all the device drivers , runs EJ0 to eject the device, and the AML causes the hardware to eject the notebook computer. The operating system then runs _STA to make sure that the docking station did indeed get out, and then the AML checks the status and returns 'not present' if the notebook was indeed successfully ejected. If the notebook was not ejected the AML will return 'present and not functioning' and deal with the problem.

~~1.1.2~~7.2.2 Warm Eject Process

In warm ejection, that is, ejection in one of the states S1-S3, the embedded controller is signaled that the system needs to eject when the system goes to sleep. The operating system refrains from acting; it first just sets a flag in the EC. When the operating system writes to the SLP_EN register, the machine goes to sleep, the hardware in the EC activates, which turns on the motors and other mechanisms that eject the notebook. Some time later (duration to be determined by the OEM), the system wakes up and the operating system will run _STA to make sure that the docking station was indeed ejected. It returns '0' if it did get out, and the message, 'Present but not working' if it did not.



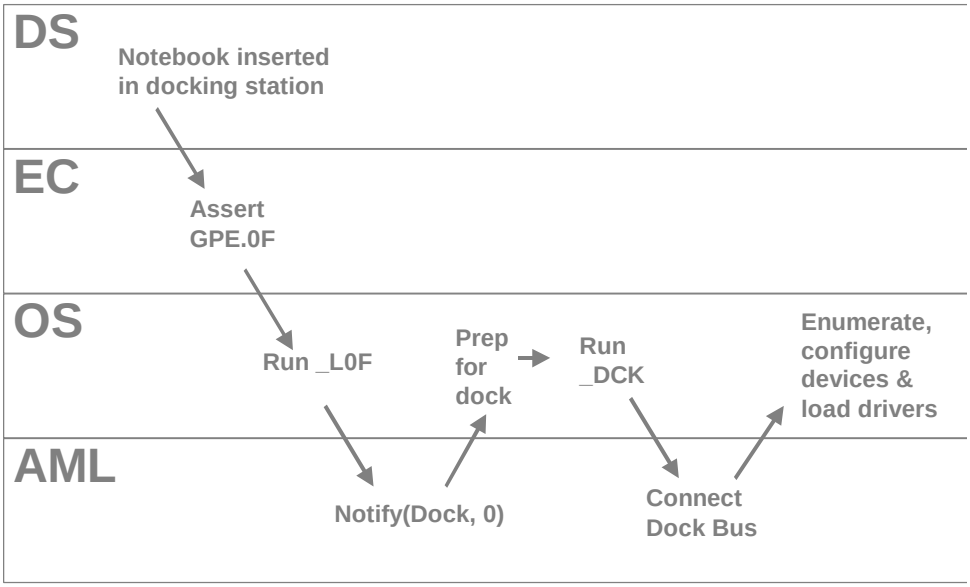
Warm Eject Process



Warm Eject Process (continued)

~~1.3~~7.3 Docking

When you dock your machine, it needs to assert a GPE. The Lxx control method loads any AML it needs to out of the docking station, using the load operator. The Lxx then issues a Notify with a code of '0' on the docking station that says, in effect, to go looking for new devices; this is Notify(dock, 0). Before this Notify is issued, you need to have all the AML. All the _STA objects for anything on the dock must not return 'Present' until _DCK is run.



The Docking Process

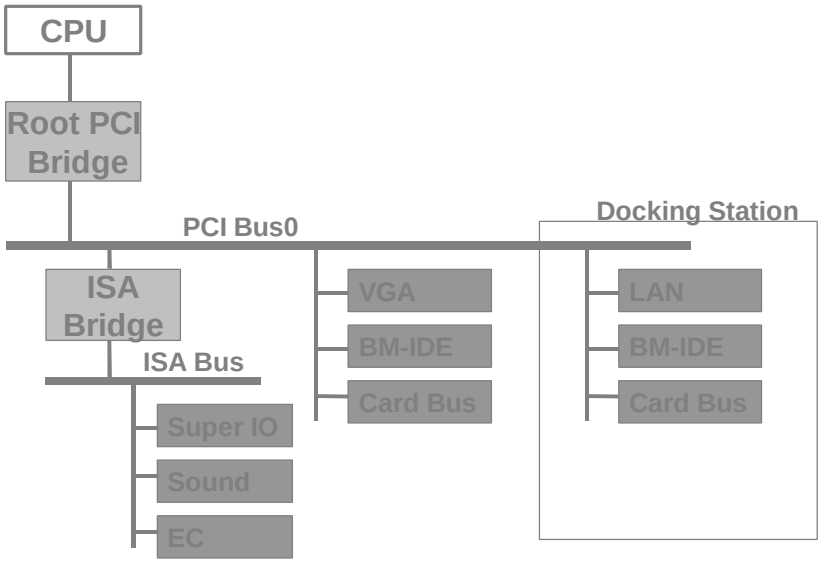
After the Notify is received and all preparatory work is complete, the operating system will run _DCK to open up the bus, and then begin the process of enumeration of the devices, configuring them. The operating system is responsible for the configuration of all the devices in the docking station. Then the device drivers are loaded.

~~1.1.1~~7.3.1 Loading of Device Drivers

The process of loading the device drivers is similar. The notebook is inserted, asserts a GPE (in this example GPE.0F is used instead of GPE.0E), the operating system runs _L0F. The AML sends a Notify(Dock, 0), and the operating system prepares to dock, running _DCK. Then the dock bus connection takes place, and the operating system begins enumerating and configurating device drivers. After all these operations, the machine is docked.

~~1.4~~7.4 A Different Implementation for a Docking Station

The preceding example represents one way of docking—an implementation based on a PCI docking connector. Another implementation involves passing the PCI bus straight through the docking connector, so that Bus 0 goes into the docking station and you have some PCI devices sitting in your docking station. In the following diagram, you will notice the PCI devices include a net card, a bus master IDE card, and a cardbus card.



Another Implementation

To properly configure such a system, you must first pick a device that is going to represent your docking station. Since this scenario does not include a desirable docking bridge, this implementation is not as straightforward as the previous one described in this chapter. You must choose something to serve as your docking bridge. In this example, the LAN adapter is chosen for this role, as the following ASL code illustrates. Any of the other devices in the docking station could also have been chosen.

```
Scope(\_SB.PCI0) {
      Device(LAN) {
            Name(_ADR, …)
            Method(_EJ0, 0) {…}
            Method(_DCK, 1) {…}
            Name(_BDN, …)
            Method(_STA, 0) {…}
}
```

In the ASL, the LAN adapter must be described as a docking station.

As you might expect, this scenario requires an EJ0, a _DCK, a _BDN, and an _STA. Since a netcard is obviously not a docking station, you need something to “hang” all these objects on. Use the LAN adapter. Now, to express the fact that there exist these other devices that are in the docking station but not hierarchically *below* the LAN adapter, you must use EJD, and this is consequently how you describe those devices. For example,

```
Scope(\_SB.PCI0) {
      Device(IDE2) {
            Name(_ADR, …)
            Name(_EJD, \_SB.PCI0.LAN)
}
      Device(CB2) {
            Name(_ADR, …)
            Name(_EJD, \_SB.PCI0.LAN)
}
```

In the preceding ASL code, you can see the IDE connector and the Card Bus connector with EJDs that refer back to the net card. This describes the docking station, that is, its existence.

Notice that the notifications for dock and undock are exactly the same as in the previous example, the only difference being that we are pointing out the LAN adapter and performing all of the notifications on that device rather than the PCI bridge, which is not present.

~~1.5~~7.5 Additional Notes: _EJx Control Methods versus _LCK Control Methods

The eject control methods are for "VCR-style" ejection. _EJx supports all ejection scenarios: hot ejection, warm ejection, and cold ejection, whereas _LCK only supports hot ejection. EJx is really designed for single devices and docking stations. _LCK is intended for single devices. For example, if someone is designing a warm "dock-able" docking station that does not have a VCR-style ejection mechanism, they should use EJx.

In summary, to implement docking, you will need to set the docking capabilities in your ASL code. You need Lxx methods to handle docking, including one to handle your eject button, if you have one. And if you have any non-hierarchical dependencies, you will need an EJD.

- **Set DCK_CAP in FACP**
- **Choose a device as the docking device**
    - _EJx - to eject notebook
    - _DCK - to identify and control dock
    - _BDN - (optional) to indicate PNPBIOS ID
    - _STA - indicates status
- **Create _Lxx method to handle docking**
    - Use Notify(dock, 0)
- **Create _Lxx method to handle eject button**
    - Use Notify(dock, 1)
- **Describe non-hierarchical dependencies**
- **Use _EJD to point to docking device**

Docking Design Checklist

8. ACPI BIOS Case Study

This section uses the ACPI-compatible Trajan 430TX motherboard as a case study and shows how to build an ACPI BIOS for an actual ACPI-compatible motherboard product. The Trajan 430TX is available for customers for evaluation along with source code for the following functionality:

- Chipset start up
    - Memory sizing
    - Chipset register initialization
    - Cache initialization
- Legacy power management functions
- Legacy docking functions
- SMBus functions

Besides describing the functionality listed above (with ACPI name space and ASL code examples), this section of the *Guide* focuses on the role of an ACPI BIOS during system state transitions:

- During the cold boot sequence.
- Waking up from S1
- Waking up from S2
- Waking up from S3
- Waking up from S4
- Switching between ACPI and Legacy modes

This section also focuses on how ACPI can be used on a Trajan motherboard to accomplish

- Power management
- Device Plug and Play

8.1 Trajan Architecture

Features of the Trajan architecture are:

- Intel 430 TX chipset with a P55C processor.
- EIO bus supporting the following components:
    - BIOS ROM
    - ISA slots
    - National 87338 Super I/O chip
    - ESS 1888 Sound device
- SMBus supporting an embedded controller interface to a smart battery
- Thermal management support using a PIIX-4 interface to a National LM-75 Temperature Controller
- A PCI root bus with the following components attached:
    - TI 1130 CardBus Controller
    - HDD Controllers
    - PCI slots
    - Chips & Technologies 65554 video controller
- A docking connector for the Proteus II (Moon II)

8.1.1 Modeling the Trajan Motherboard with Objects in the ACPI Namespace

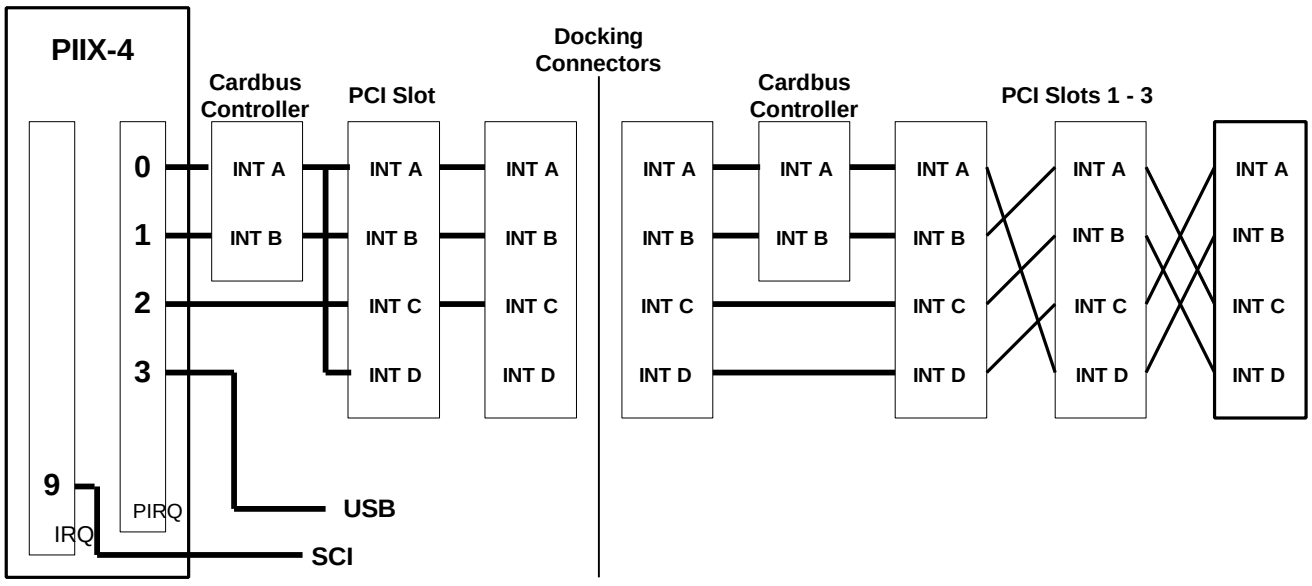
An ACPI name space of device objects that models (at an abstract level) the Trajan motherboard is shown below (the device object names shown below are used in ASL example code later in this section). Devices not represented by objects in the name space below are

- The SMBus devices, particularly the ACPI EC interface-compatible embedded controller (EC) that interfaces with the Smart Battery system of a charger and dual batteries.
- The temperature device (LM-75 accessed through PIIX-4 using ACPI control methods).

```
\_PR
      CPU0
\PIDE                              //Power resource for IDE0
\PUA1                              //Power resource for UART
\_TZ                         //Thermal control
      PFAN                         //Power resource for Fan
      FAN                          //FAN Device
      THRM                         //Thermal zone
\_GPE                              //General Purpose event
      _L0B                         //Lid event
      _L09                   //Docking event
      _L00                   //Thermal event
\_SB
      PCI0                   //PCI root bridge
            PX40                   //PCI-ISA Bridge
                  EIO              //EIO bus
                        SIO1       //SIO devices (Floppy, UART1, UART2, LPT)
                        MDEV       //Motherboard devices (Keyboard, Mouse, . . .)
            PX41                   //PCI IDE Controller
            PX42                   //USB Host Controller
            PX43                   //Power Management Controller
            MPCI                   //Docking PCI to PCI bridge
                  DIDE             //Docking PCI IDE Controller
                  DCBC             //Docking CardBus Controller
                  MISA             //Docking PCI to ISA bus
                        ISA        //Docking ISA bus
                              DBRD //Docking Board Devices
                              SIO2 //Docking SIO Devices (Floppy, UART2)
```

~~1.1.2~~8.1.2 Trajan Interrupt Structure

The interrupt structure of the Trajan is shown in the following diagram:



The interrupt structure to the left of the line labeled "Docking Connectors" in the diagram can be modeled as a PCI interrupt routing table using ASL code as shown below.

```
Scope(\_SB) {
    // PCI interrupt link
    Device(LNKA) {
        Name(_HID, EISAID("PNP0C0F"))
        Name(_UID, 1)
        Method(_STA, 0) {
            And(PIRA, 0x80, Local0)          // Get PIRQ routing
            If(LEqual(Local0, 0x80)) {
                Return(0x09)                 // Present, but disabled
            }
            Else {
                Return(0x0F)                 // Present, enabled, functioning
            }
        }


        Name(_PRS, ResourceTemplate(){
                    // IRQ descriptor, level, low, IRQ3-7,9-12,14,15
                    IRQ(Level, ActiveLow, Shared) {
                        3,4,5,6,7,9,10,11,12,14,15}
        })

        Method(_DIS) {
            // Disable PIRQ routing
            Or(PIRA, 0x80, PIRA)
        }

        Method(_CRS) {
            Name(BUFA, Buffer(){
                    // IRQ descriptor, level, low, IRQ3-7,9-12,14,15
                    0x23, 0x00, 0x00, 0x18,
                    // End tag
                    0x79, 0})
            CreateByteField(BUFA, 0x01, IRA1)           // IRQ mask 1
            CreateByteField(BUFA, 0x02, IRA2)           // IRQ mask 2
            Store(0, Local3)                            // Init local variables
            Store(0, Local4)
            And(PIRA, 0x8F, Local0)                     // Get register value
            If(LLess(Local0, 0x80)) {                   // Routing enable
                And(Local0, 0x0F, Local0)               // Mask off enable bit
                If(LGreater(Local0, 0x07)) {            // For upper byte mask
                    Subtract(Local0, 8, Local2)         // Adjust it
                    ShiftLeft(One, Local2, Local4)}     // Convert to bit mask
                Else {If(LGreater(Local0, 0)) {         // For lower byte mask
                    ShiftLeft(One, Local0, Local3)}}    // Convert to bit mask
                Store(Local3, IRA1)                     // Update buffer
                Store(Local4, IRA2)
            }
            Return(BUFA)
        } // Method(_CRS)

        Method(_SRS, 1) {
            // ARG0 = PnP resource string to set
            // (IRQ ordering assumed to be same as _PRS/_CRS)
            CreateByteField(ARG0, 0x01, IRA1)           // IRQ mask 1
            CreateByteField(ARG0, 0x02, IRA2)           // IRQ mask 2

            ShiftLeft(IRA2, 8, Local0)                  // Shift into upper byte
            Or(Local0, IRA1, Local0)                    // Convert to word

            // Local0 should have only one bit set here!!!
            Store(0, Local1)                            // Init local variable
            ShiftRight(Local0, 1, Local0)               // Adjust for zero based
            While(LGreater(Local0, 0)) {                // Convert bit mask to number
                Increment(Local1)
                ShiftRight(Local0, 1, Local0)
            }

            // Enable and set PIRQ routing
            And(PIRA, 0x70, Local0)
            Or(Local1, Local0, PIRA)
```

```
        } // Method(_SRS, 1)
    } // Device(LNKA)
//
    Device(LNKB) {
        Name(_HID, EISAID("PNP0C0F"))
        Name(_UID, 2)
        Method(_STA, 0) {
            And(PIRB, 0x80, Local0)          // Get PIRQ routing
            If(LEqual(Local0, 0x80)) {
                Return(0x09)                 // Present, but disabled
            }
            Else {
                Return(0x0F)                 // Present, enabled, functioning
            }
        }


        Name(_PRS, ResourceTemplate(){
                    // IRQ descriptor, level, low, IRQ3-7,9-12,14,15
                    IRQ(Level, ActiveLow, Shared) {
                        3,4,5,6,7,9,10,11,12,14,15}
        })

        Method(_DIS) {
            // Disable PIRQ routing
            Or(PIRB, 0x80, PIRB)
        }

        Method(_CRS) {
            Name(BUFB, Buffer(){
                    // IRQ descriptor, level, low, IRQ3-7,9-12,14,15
                    0x23, 0x00, 0x00, 0x18,
                    // End tag
                    0x79, 0})
            CreateByteField(BUFB, 0x01, IRB1)           // IRQ mask 1
            CreateByteField(BUFB, 0x02, IRB2)           // IRQ mask 2
            Store(0, Local3)                            // Init local variables
            Store(0, Local4)
            And(PIRB, 0x8F, Local0)                     // Get register value
            If(LLess(Local0, 0x80)) {                   // Routing enable
                And(Local0, 0x0F, Local0)               // Mask off enable bit
                If(LGreater(Local0, 0x07)) {            // For upper byte mask
                    Subtract(Local0, 8, Local2)         // Adjust it
                    ShiftLeft(One, Local2, Local4)}     // Convert to bit mask
                Else {If(LGreater(Local0, 0)) {         // For lower byte mask
                    ShiftLeft(One, Local0, Local3)}}    // Convert to bit mask
                Store(Local3, IRB1)                     // Update buffer
                Store(Local4, IRB2)
            }
            Return(BUFB)
        } // Method(_CRS)

        Method(_SRS, 1) {
            // ARG0 = PnP resource string to set
            // (IRQ ordering assumed to be same as _PRS/_CRS)
            CreateByteField(ARG0, 0x01, IRB1)           // IRQ mask 1
            CreateByteField(ARG0, 0x02, IRB2)           // IRQ mask 2

            ShiftLeft(IRB2, 8, Local0)                  // Shift into upper byte
            Or(Local0, IRB1, Local0)                    // Convert to word

            // Local0 should have only one bit set here!!!
            Store(0, Local1)                            // Init local variable
            ShiftRight(Local0, 1, Local0)               // Adjust for zero based
            While(LGreater(Local0, 0)) {                // Convert bit mask to number
                Increment(Local1)
                ShiftRight(Local0, 1, Local0)
            }

            // Enable and set PIRQ routing
            And(PIRB, 0x70, Local0)
```

```
                Or(Local1, Local0, PIRB)
            } // Method(_SRS, 1)
        } // Device(LNKB)
//
        Device(LNKC) {
            Name(_HID, EISAID("PNP0C0F"))
            Name(_UID, 3)
            Method(_STA, 0) {
                And(PIRC, 0x80, Local0)         // Get PIRQ routing
                If(LEqual(Local0, 0x80)) {
                    Return(0x09)                    // Present, but disabled
                }
                Else {
                    Return(0x0F)                    // Present, enabled, functioning
                }
            }

            Name(_PRS, ResourceTemplate(){
                        // IRQ descriptor, level, low, IRQ3-7,9-12,14,15
                        IRQ(Level, ActiveLow, Shared) {
                            3,4,5,6,7,9,10,11,12,14,15}
            })

            Method(_DIS) {
                // Disable PIRQ routing
                Or(PIRC, 0x80, PIRC)
            }

            Method(_CRS) {
                Name(BUFC, Buffer(){
                        // IRQ descriptor, level, low, IRQ3-7,9-12,14,15
                        0x23, 0x00, 0x00, 0x18,
                        // End tag
                        0x79, 0})
                CreateByteField(BUFC, 0x01, IRC1)                   // IRQ mask 1
                CreateByteField(BUFC, 0x02, IRC2)                   // IRQ mask 2
                Store(0, Local3)                                    // Init local variables
                Store(0, Local4)
                And(PIRC, 0x8F, Local0)                             // Get register value
                If(LLess(Local0, 0x80)) {                           // Routing enable
                    And(Local0, 0x0F, Local0)                       // Mask off enable bit
                    If(LGreater(Local0, 0x07)) {                    // For upper byte mask
                        Subtract(Local0, 8, Local2)             // Adjust it
                        ShiftLeft(One, Local2, Local4)}         // Convert to bit mask
                    Else {If(LGreater(Local0, 0)) {             // For lower byte mask
                        ShiftLeft(One, Local0, Local3)}}        // Convert to bit mask
                    Store(Local3, IRC1)                             // Update buffer
                    Store(Local4, IRC2)
                }
                Return(BUFC)
            } // Method(_CRS)

            Method(_SRS, 1) {
                // ARG0 = PnP resource string to set
                // (IRQ ordering assumed to be same as _PRS/_CRS)
                CreateByteField(ARG0, 0x01, IRC1)                   // IRQ mask 1
                CreateByteField(ARG0, 0x02, IRC2)                   // IRQ mask 2

                ShiftLeft(IRC2, 8, Local0)                          // Shift into upper byte
                Or(Local0, IRC1, Local0)                            // Convert to word

                // Local0 should have only one bit set here!!!
                Store(0, Local1)                                    // Init local variable
                ShiftRight(Local0, 1, Local0)                       // Adjust for zero based
                While(LGreater(Local0, 0)) {                        // Convert bit mask to number
                    Increment(Local1)
                    ShiftRight(Local0, 1, Local0)
                }

                // Enable and set PIRQ routing
                And(PIRC, 0x70, Local0)
```

```
                Or(Local1, Local0, PIRC)
            } // Method(_SRS, 1)
        } // Device(LNKC)
//
        Device(LNKD) {
            Name(_HID, EISAID("PNP0C0F"))
            Name(_UID, 4)
            Method(_STA, 0) {
                And(PIRD, 0x80, Local0)         // Get PIRQ routing
                If(LEqual(Local0, 0x80)) {
                    Return(0x09)                    // Present, but disabled
                }
                Else {
                    Return(0x0F)                    // Present, enabled, functioning
                }
            }

            Name(_PRS, ResourceTemplate(){
                        // IRQ descriptor, level, low, IRQ3-7,9-12,14,15
                        IRQ(Level, ActiveLow, Shared) {
                            3,4,5,6,7,9,10,11,12,14,15}
            })

            Method(_DIS) {
                // Disable PIRQ routing
                Or(PIRD, 0x80, PIRD)
            }

            Method(_CRS) {
                Name(BUFD, Buffer(){
                        // IRQ descriptor, level, low, IRQ3-7,9-12,14,15
                        0x23, 0x00, 0x00, 0x18,
                        // End tag
                        0x79, 0})
                CreateByteField(BUFD, 0x01, IRD1)                   // IRQ mask 1
                CreateByteField(BUFD, 0x02, IRD2)                   // IRQ mask 2
                Store(0, Local3)                                    // Init local variables
                Store(0, Local4)
                And(PIRD, 0x8F, Local0)                             // Get register value
                If(LLess(Local0, 0x80)) {                           // Routing enable
                    And(Local0, 0x0F, Local0)                       // Mask off enable bit
                    If(LGreater(Local0, 0x07)) {                    // For upper byte mask
                        Subtract(Local0, 8, Local2)                 // Adjust it
                        ShiftLeft(One, Local2, Local4)}             // Convert to bit mask
                    Else {If(LGreater(Local0, 0)) {                 //For lower byte mask
                        ShiftLeft(One, Local0, Local3)}}            // Convert to bit mask
                    Store(Local3, IRD1)                             // Update buffer
                    Store(Local4, IRD2)
                }
                Return(BUFD)
            } // Method(_CRS)

            Method(_SRS, 1) {
                // ARG0 = PnP resource string to set
                // (IRQ ordering assumed to be same as _PRS/_CRS)
                CreateByteField(ARG0, 0x01, IRD1)                   // IRQ mask 1
                CreateByteField(ARG0, 0x02, IRD2)                   // IRQ mask 2

                ShiftLeft(IRD2, 8, Local0)                          // Shift into upper byte
                Or(Local0, IRD1, Local0)                            // Convert to word

                // Local0 should have only one bit set here!!!
                Store(0, Local1)                                    // Init local variable
                ShiftRight(Local0, 1, Local0)                       // Adjust for zero based
                While(LGreater(Local0, 0)) {                        // Convert bit mask to number
                    Increment(Local1)
                    ShiftRight(Local0, 1, Local0)
                }

                // Enable and set PIRQ routing
                And(PIRD, 0x70, Local0)
```

```
            Or(Local1, Local0, PIRD)
        } // Method(_SRS, 1)
} // Device(LNKD)
        .
        .
        .
// PCI Bus/Device
Device(PCI0) {
        Name(_HID, EISAID("PNP0A03"))           // PCI root bus ID
        Name(_ADR, 0x00000000)                  // word dev/func = 0/0
        Name(_CRS, 0)                           // bus 0 (for root bus only)
        Name(_PRT, Package(){
                                Package(){0x0001FFFF, 0, LNKA, 0},      // Chipset
                                Package(){0x0001FFFF, 1, LNKB, 0},
                                Package(){0x0001FFFF, 2, LNKC, 0},
                                Package(){0x0001FFFF, 3, LNKD, 0},
                                Package(){0x0004FFFF, 0, LNKA, 0}, // PCI slot
                                Package(){0x0004FFFF, 1, LNKB, 0},
                                Package(){0x0004FFFF, 2, LNKC, 0},
                                Package(){0x0004FFFF, 3, LNKA, 0},
                                Package(){0x000AFFFF, 0, LNKA, 0},// Cardbus
                                Package(){0x000AFFFF, 1, LNKB, 0}
                                })
                .
                .
                .
        Device(PX40) {
                Name(_ADR, 0x00010000)                                  // word dev/func = 1/0
                OperationRegion(PIRQ, PCI_Config, 0x60, 0x04)
                Scope(\) {
                        Field(\_SB.PCI0.PX40.PIRQ, ByteAcc, NoLock, Preserve) {
                                PIRA, 8,
                                PIRB, 8,
                                PIRC, 8,
                                PIRD, 8
                        }
                }
                .
                .
                .
```

~~1.1.3~~8.1.3 Trajan Thermal Control

The Trajan platform monitors temperature using the National Semiconductor LM75. The LM75 SMBus is connected to the PIIX-4 SMBus. The thermal output line is connected to “Therm” on the PIIX-4. This section describes how to implement a solution that emulates multiple thermal zones using periodic software that reads the thermal sensors.

A legacy power management timer is programmed to generate an SMI at some fixed interval. The SMI handler will then poll the LM75 and see if any threshold has been crossed. If a threshold has been crossed, the SMI handler will cause an SCI to be generated to notify the OS that a thermal event has occurred.

Step 1:

During BIOS POST, the LM75 is initialized with maximum value for Tos and minimum value for Thyst; this effectively prevents the LM75 from generating any signal on its own. The following sample code is one way to do this:

```
SetTempRegs:
        mov  cx, 0C900h                 ; Set Thyst to min
        mov  bx, 07D00h                 ; Set Tos to max

        ; Set LM75 Tos register
        WriteSmbRtReg 003h, 003h        ; Select Tos register
        WriteSmbRtReg 004h, 090h        ; ATF smb address = 90h + write
        ; PIIX4 sends out word data from DAT0 register (05h) first.
        ; But the ATF (LM75) expects high byte first.
        ; So put high byte into DAT0 (05h) and low byte into DAT1 (06h)
        WriteSmbRtReg 006h, <bl>        ; Low byte data
        WriteSmbRtReg 005h, <bh>        ; High byte data
        WriteSmbRtReg 002h, 04Ch        ; Issue "word data" command
        smbPiix4Wait                    ; Wait for command to complete

        ; Set LM75 Thyst register
        WriteSmbRtReg 003h, 002h        ; Select Thyst register
        ; PIIX4 sends out word data from DAT0 register (05h) first.
        ; But the ATF (LM75) expects high byte first.
        ; So put high byte into DAT0 (05h) and low byte into DAT1 (06h)
        WriteSmbRtReg 006h, <cl>        ; Low byte data
        WriteSmbRtReg 005h, <ch>        ; High byte data
        WriteSmbRtReg 002h, 04Ch        ; Issue "word data" command
        smbPiix4Wait                    ; Wait for command to complete
```

Note that an alternative method is to initialize Tos with the critical threshold temperature which would allow the LM75 to generate a signal when the critical threshold is crossed.

Step 2:

When the OS switches the system to ACPI mode through the SMI_CMD port, the SMI ACPI enabling routine will enable and program a legacy timer with a fixed interval. When the timer counts down to zero, a SCI is generated. Sample code that enables and programs a legacy timer with a fixed interval is shown below.

```
; Enable SW timer for thermal purposes. The SW timer is used to
; periodically poll the LM75 to monitor temperature changes.
; This solution is needed in order to monitor more than one temperature zone on an
; ACPI system equipped with one LM75.
; Select 8-second granularity for SW timer and set count to 1 unit
        ChangeRegister16 044h, NOT 0F07Fh, 00100h, PIIX4_PM_DEV_NUM
        WritePmRtReg32 02Ch, 00000h     ; Toggle to load SW timer
        WritePmRtReg32 02Ch, 00040h
```

Note that if shorter polling period is required, a different timer could be selected.

Step 3:

When the timer SMI is generated, the SMI handler will poll the LM75 for the current temperature. If the temperature has crossed any threshold, the thermal polarity bit of the chipset can be toggled to latch a thermal event on the GP_STS register. When the system returns from SMI mode, the SCI request will be generated. Sample SMI handler code is shown below.

```
; Thermal Device (LM75) Constants
; Trip point 1
        TP1H    EQU 2800h               ; 40c
        TP1L    EQU 2600h               ; 38c
; Trip point 2
        TP2H    EQU 2E00h               ; 46c
        TP2L    EQU 2C00h               ; 44c
; Critical trip point
        TP3H    EQU 3200h               ; 50c
        TP3L    EQU 3100h               ; 49c

        ChangePmRtReg32 02Ch, NOT 000000040h, 000000000h ; Toggle enable bit to reload timer
        ChangePmRtReg32 02Ch, NOT 000000040h, 000000040h

        call    readCurrTemp    ; Get current temperature
        push ax                         ; Save a copy
        cmp  ax, LastTemp               ; Compare with last temperature to find out
        jae     TempUp                          ; if temperature is going up or down

TempDown:                               ; Temperature is going down
        cmp  LastTemp, TP3L             ; Check if last temperature was below zone 3
        jb      CheckDownZone2          ; If not
        cmp  ax, TP3L                   ; Is it below zone 3 now?
        jb      PassThreshold           ; If yes, go generate SCI condition
        jmp  DonePollTemp               ; If not, nothing to do

CheckDownZone2:                         ; Check zone 2 if last temperature was below zone 3
        cmp  LastTemp, TP2L             ; Check if last temperature was below zone 2
        jb      CheckDownZone1          ; If not
        cmp  ax, TP2L                   ; It is below zone 2 now?
        jb      PassThreshold           ; If yes, go generate SCI condition
        jmp  DonePollTemp               ; If not, nothing to do

CheckDownZone1:                         ; Check zone 1 if last temperature was below zone 2
        cmp  LastTemp, TP1L             ; Check if last temperature was below zone 1
        jb      DonePollTemp            ; If not
        cmp  ax, TP1L                   ; It is below zone 1 now?
        jb      PassThreshold           ; If yes, go generate SCI condition
        jmp  DonePollTemp               ; If not, nothing to do

TempUp:                                 ; Temperature is going up
        cmp  LastTemp, TP1H             ; Check if last temperature was above zone 1
        ja      CheckUpZone2            ; If not
        cmp  ax, TP1H                   ; Is it above zone 1 now?
        ja      PassThreshold           ; If yes, go generate SCI condition
        jmp  DonePollTemp               ; If not, nothing to do

CheckUpZone2:                           ; Check zone 2 if last temperature was above zone 1
        cmp  LastTemp, TP2H             ; Check if last temperature was above zone 2
        ja      CheckUpZone3            ; If not
        cmp  ax, TP2H                   ; Is it above zone 2 now?
        ja      PassThreshold           ; If yes, go generate SCI condition
        jmp  DonePollTemp               ; If not, nothing to do

CheckUpZone3:                           ; Check zone 3 if last temperature was above zone 2
        cmp  ax, TP3H                   ; Go generate SCI condition if temperature is above zone 3
        ja      PassThreshold
        jmp  DonePollTemp

PassThreshold:                          ; Threshold is crossed since last temperature was read
        ReadPmRtReg 028h                ; Toggle thermal polarity bit
                                                ; to set thermal event status bit
        xor  al, 004h
        WritePmRtReg 028h, al
        xor  al, 004h
        WritePmRtReg 028h, al

DonePollTemp:
        pop  ax
        mov  LastTemp, ax               ; Update last temperature
```

```
pop   bx
```

~~1.2~~8.2 Initializing the ACPI BIOS During POST and Cold Boot Sequence

Initializing the ACPI BIOS is one step in the cold boot sequence of the Trajan. The steps in the cold boot sequence are shown below; initializing the ACPI BIOS is the next to the last step:

1. Memory testing and configuration.
2. BIOS shadow.
3. Minimal test of motherboard devices.
4. Identify PnP ISA devices.
5. Allocate resources to static devices.
6. Enable I/O devices and PnP PCI.
7. Configure the IPL device.
8. Enable ISA PnP.
9. Initialize legacy power management.
10. Initialize the ACPI portion of the BIOS.
11. Int19

The next to the last step in the cold boot sequence, initializing the ACPI BIOS, is made up of the following steps:

1. Build the ACPI tables.
2. Update and adjust the DOS INT15h memory routines.
3. Initialize the chipset registers.
4. Save the chipset/configuration data.
5. Calculate the hardware signature.

8.2.1 Building the ACPI Tables in Memory

This section

- Describes the process the Trajan BIOS uses to load the ACPI tables into memory and shows the location of the various ACPI tables in memory after this process is done.
- Describes the logical structure of the ACPI tables in memory.
- Details the contents of the Trajan FACP table.

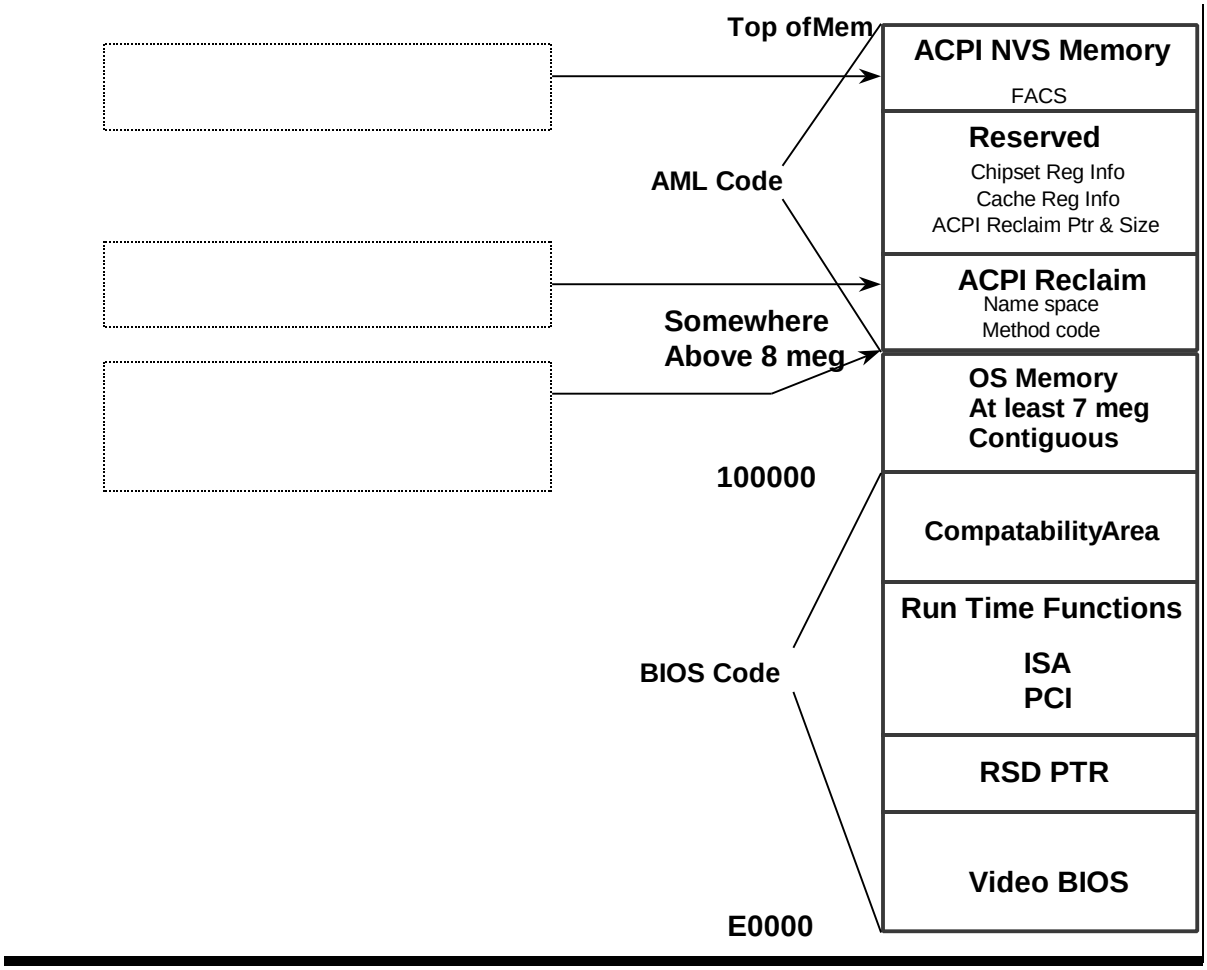
8.2.2 Sizing Memory, Allocating Memory, Fixing Up Table Pointers, and Copying ACPI Tables into Memory

The process the Trajan BIOS uses to copy the ACPI tables into memory is:

1. The BIOS determines the size of physical memory on the platform by using existing BIOS functions and standard CMOS to determine the extended memory size and the memory address at the top of memory.
2. The BIOS allocates the ACPI Reserved area at the top of memory, and within that allocates the ACPI NVS and ACPI Reclaim memory areas. The NVS memory area must include room for the FACS table and the chipset register data. The Reclaim area must include space for RSDT, FACP, and DSDT tables; about 16K is allocated for the DSDT table.
3. The BIOS uses the allocated physical memory addresses to fix up the ACPI table pointers with physical address values, calculates the table checksum values (now that all data values are in the tables except checksums) and stores these checksum values in the tables.

4. The BIOS copies the FACS table into the NVS portion of the ACPI Reserved memory area; copies the RSDT, FACP, and DSDT into the ACPI Reclaim memory area, and copies the RSD Pointer into the E000:F000 memory segment.

After the BIOS is done copying the ACPI tables into memory, the resulting memory map is shown in the illustration below.

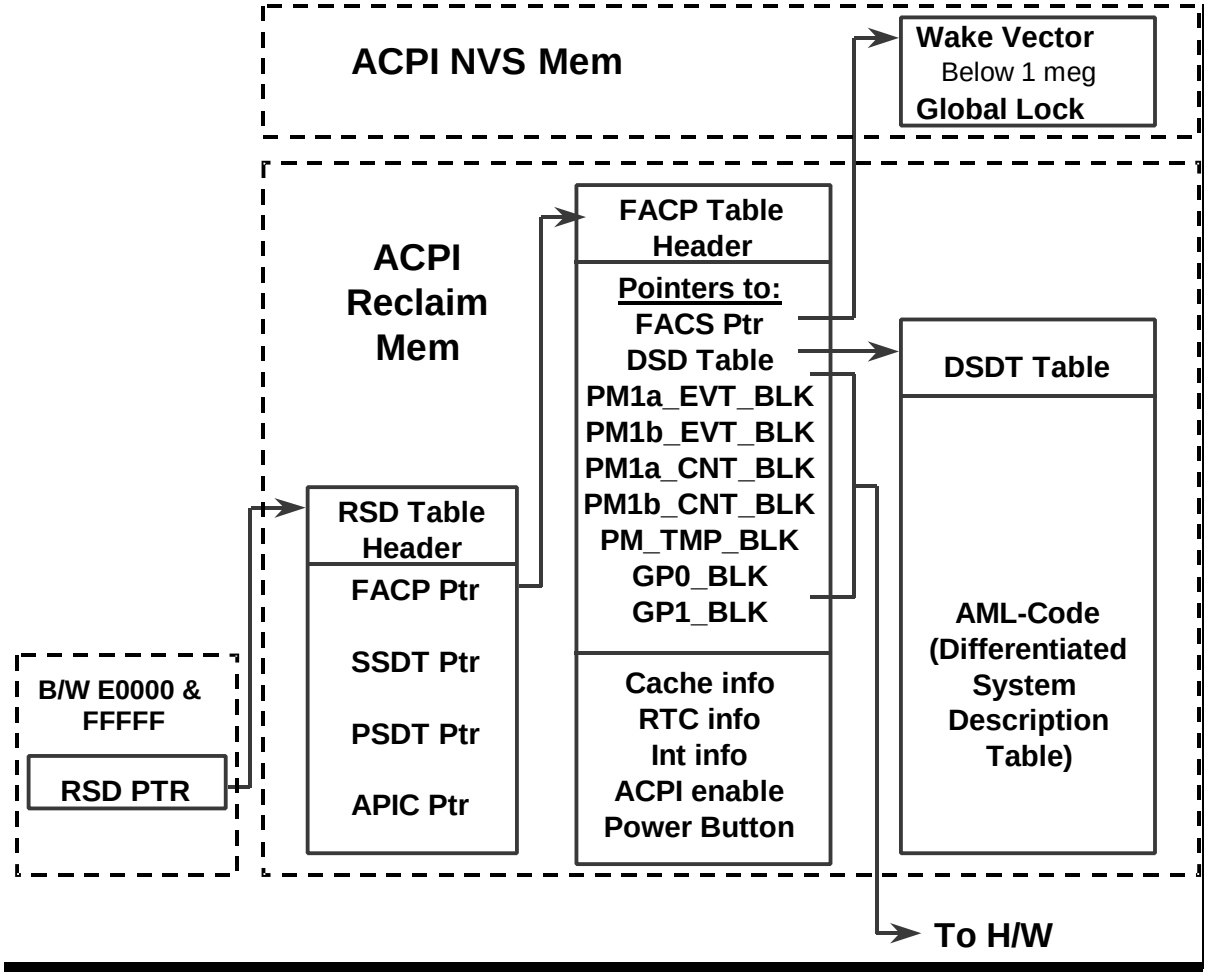


The annotations on the above illustration point out the DOS Int 15H functions the BIOS can use determine the size and location of various types of memory:

DOS Int15H Function	Description	Comment
Function 88	Reports up to 16 MB of memory, but not ACPI Reserved memory. For example, on a machine with 16MB of RAM, if 1MB is dedicated to ACPI tables, this function will return 15 MB of memory.	Prevents compatibility problems with old memory managers.
Function E801	Can report more than 16 MB of memory, but not ACPI Reserved memory. For example, on a machine with 32 MB of RAM, if 1 MB is dedicated to ACPI	Prevents compatibility problems with old memory managers.

	tables, this function will return 31 MB of memory.	
Function E820	Reports memory maps and reserved areas. For example, on a machine with 64 MB of RAM, if 1MB is dedicated to ACPI tables, this function will return 64 MB of memory.	Expanded to report ACPI reserved areas.

The logical structure of the loaded ACPI tables with the fixed up pointers is shown in the following figure:



The values in the FACP table for the Trajan with 16 MB of DRAM are shown in the following table.

Field	Byte Length	Trajan Value	Comment
Signature	4	'FACP'	Must be 'FACP.'
Length	4	74h	Length, in bytes, of the entire Fixed ACPI Description Table.
Revision	1	0x01	For Revision 1.0 of the ACPI Specification, must be 0x01.

Field	Byte Length	Trajan Value	Comment
Checksum	1	0x96	Entire table must sum to zero.
OEMID	6	'Intel'	
OEM Table ID	8	'Trajan'	Manufacturer model ID.
OEM Revision	4	0x1000	OEM DSDT revision.
Creator ID	4	0	Vendor ID of utility that created the table. For the DSDT, RSDT, SSDT, and PSDT tables, this is the ID for the ASL Compiler.
Creator Revision	4	0	Revision of utility that created the table. For the DSDT, RSDT, SSDT, PSDT tables, this is the revision for the ASL Compiler.
FIRMWARE_CTRL	4	0xFFFF80	Physical memory address (0-4 GB) of the Firmware ACPI Control Structure (FACS).
DSDT	4	0xFFBA30	Physical memory address (0-4 GB) of the Differentiated System Description Table (DSDT).
INT_MODEL	1	0	This value represents the interrupt model being assumed in the ACPI description of the OS. This value therefore represents the interrupt model. This value is not allowed to change for a given machine, even across reboots. 0 = Dual PIC, industry standard PC-AT type implementation with 0-15 IRQs with EISA edge-level-control register.
SCI_INT	2	0x09	System pin the SCI interrupt is wired to. The OS is required to treat the ACPI SCI interrupt as a sharable, level, active low interrupt.
SMI_CMD	4	0xB2	System port address of the SMI Command Port.
ACPI_ENABLE	1	0x2B	The value to write to SMI_CMD to disable SMI ownership of the ACPI hardware registers.
ACPI_DISABLE	1	0xD4	The value to write to SMI_CMD to re-enable SMI ownership of the ACPI hardware registers.
S4BIOS_REQ	1	0	The value to write to SMI_CMD to enter the S4BIOS state. The S4BIOS state provides an alternate way to enter the S4 state where the firmware saves and restores the memory context.
PM1a_EVT_BLK	4	0x1000	System port address of the Power Management 1a Event Register Block. This is a required field.
PM1b_EVT_BLK	4	0	System port address of the Power Management 1b Event Register Block. This field is optional; if this register block is not supported, this field contains zero.
PM1a_CNT_BLK	4	0x1004	System port address of the Power Management 1a Control Register Block. This is a required field.

Field	Byte Length	Trajan Value	Comment
PM1b_CNT_BLK	4	0	System port address of the Power Management 1b Control Register Block. This field is optional; if this register block is not supported, this field contains zero.
PM2_CNT_BLK	4	0x22	System port address of the Power Management 2 Control Register Block. This field is optional; if this register block is not supported, this field contains zero.
PM_TMR_BLK	4	0x1008	System power address of the Power Management Timer Control Register Block. This is a required field.
GPE0_BLK	4	0x100C	System port address of Generic Purpose Event 0 Register Block. This is an optional field; if this register block is not supported, this field contains zero.
GPE1_BLK	4	0	System port address of Generic Purpose Event 1 Register Block. This is an optional field; if this register block is not supported, this field contains zero.
PM1_EVT_LEN	1	4	Number of bytes in port address space decoded by PM1a_EVT_BLK. This value is ≥ 4.
PM1_CNT_LEN	1	2	Number of bytes in port address space decoded by PM1a_CNT_BLK. This value is ≥ 1.
PM2_CNT_LEN	1	1	Number of bytes in port address space decoded by PM2_CNT_BLK. This value is ≥ 1.
PM_TM_LEN	1	4	Number of bytes in port address space decoded by PM_TM_BLK. This value is ≥ 4.
GPE0_BLK_LEN	1	4	Number of bytes in port address space decoded by GPE0_BLK. The value is a non-negative multiple of 2.
GPE1_BLK_LEN	1	0	Number of bytes in port address space decoded by GPE1_BLK. The value is a non-negative multiple of 2.
GPE1_BASE	1	0	Offset within the ACPI general-purpose event model where GPE1 based events start.
P_LVL2_LAT	2	2	The worst-case hardware latency, in microseconds, to enter and exit a C2 state. A value > 100 indicates the system does not support a C2 state.
P_LVL3_LAT	2	1000	The worst-case hardware latency, in microseconds, to enter and exit a C3 state. A value > 1000 indicates the system does not support a C3 state.
FLUSH_SIZE	2	TBD	If WBINVD=0, the value of this field is the contiguous memory size that needs to be read(using cacheable addresses) to flush dirty lines from any processor's memory caches. This value is ignored if WBINVD=1.

Field	Byte Length	Trajan Value	Comment
FLUSH_STRIDE	2	TBD	If WBINVD=0, the value of this field is the memory stride width, in bytes, to perform reads to flush the processor's memory caches. This value is ignored if WBINVD=1.
DUTY_OFFSET	1	1	The zero-based index of where the processor's duty cycle setting is within the processor's P_CNT register.
DUTY_WIDTH	1	3	The bit width of the processor's duty cycle setting value in the P_CNT register.
DAY_ALRM	1	0x0D	The RTC CMOS RAM index to the day-of-month alarm value. If this field contains a zero, then the RTC day of the month alarm feature is not supported. If this field has a non-zero value, then this field contains an index into RTC RAM space that the OS can use to program the day of the month alarm.
MON_ALRM	1	0	The RTC CMOS RAM index to the month of year alarm value. If this field contains a zero, then the RTC month of the year alarm feature is not supported.
CENTURY	1	0	The RTC CMOS RAM index to the century of data value (hundred and thousand year decimals). If this field contains a zero, then the RTC centenary feature is not supported.
Flags	4	0xA4	Fixed feature flags: C1 supported; C2 supported on a UP system; power button, and RTC wakeup all implemented as fixed features; RTC S4 wakeup supported; 24-bit PM timer.

~~1.1.3~~8.2.3 Initializing the Chipset Registers

The BIOS

- Enables the chipset power management (PM) features:
  - Enables CPU clock control for LVL2 and LVL3 by writing value 12h to P_CNTRL.S10.[8-15] in the ACPI hardware register set.
  - Enables clock events for the IRQs by writing value 23h to ACTB.P58.[0-7] in the Trajan chipset Southbridge.
- Initializes the ACPI event features.
  - Clears all ACPI event status.
  - Disables all ACPI events, leaving it to the OS to later determine which events to enable.

8.2.4 Saving the Chipset /Configuration Data

The BIOS

- Saves the chipset registers:
  - Saves the POST values of the chipset registers in the ACPI Reserved memory area. Wakeup from system state S3 requires these values to restore the chipset registers.

- Saves the memory controller configuration in CMOS; in the case of the Trajan, Northbridge registers DRB.P[60-65] and DRT.P[67-68] are saved.

- Saves the ACPI table data:
  - The FACS address is saved in CMOS because the FACP table, which contains the FACS address, is not accessible after the OS reclaims memory.

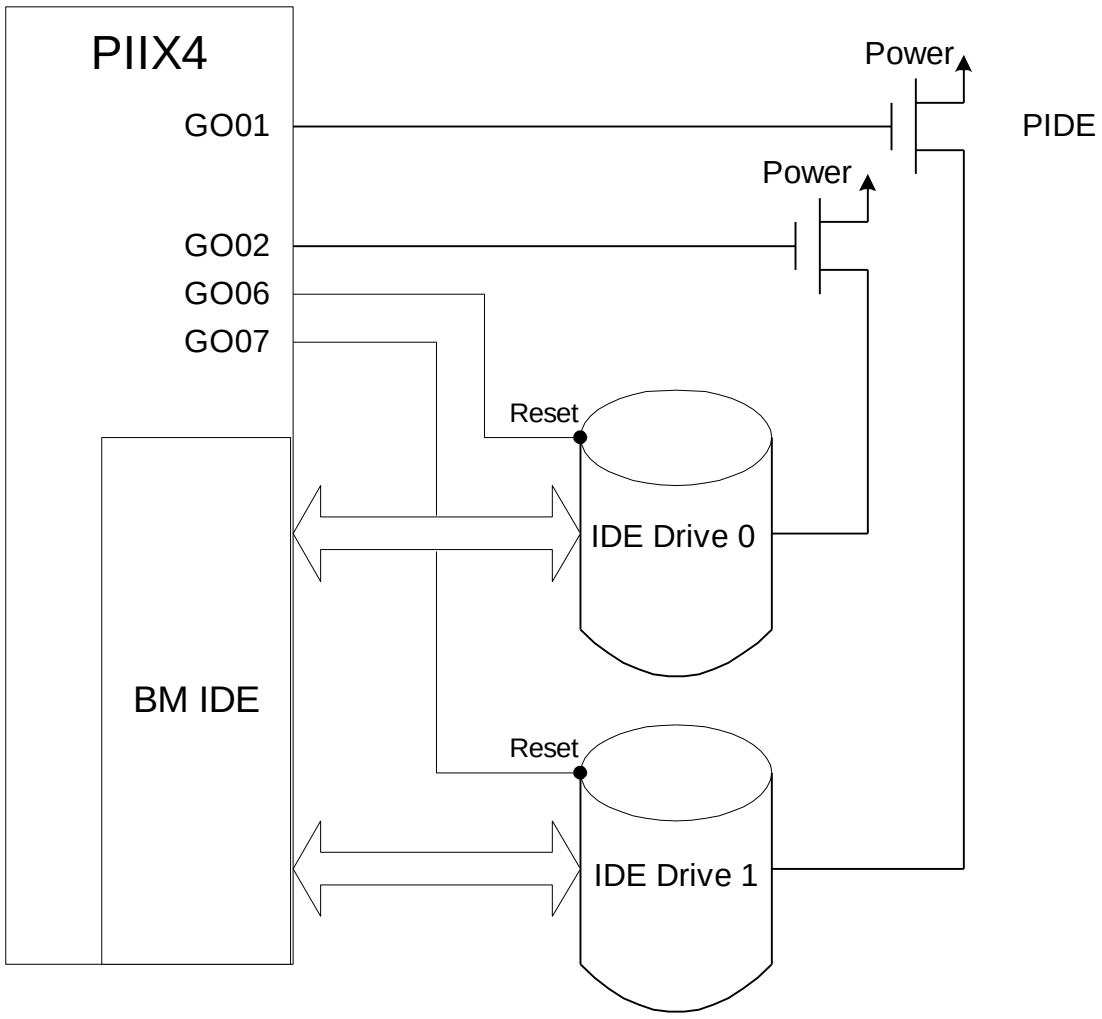
8.3 Power Management Using ACPI on the Trajan Motherboard

This section gives examples of how all three types of power management are implemented on the Trajan motherboard:

- Device power management.
- Processor power management.
- System power management.

8.3.1 Device Power Management

An ACPI PowerResource object is used to manage power on the IDE drives. The following illustration shows the relationship between pins on the PIIX-4 device on the Trajan board and the power and reset controls on a pair of IDE drives.



The following ASL code creates a PowerResource object in the Device object for IDE Drive 1:

```
PowerResource(PIDE, 0, 0) {          //Power resource for IDE Drive 1
                                          //SystemLevel parameter = 0, which means
                                          //this IDE drive can be turned off in
                                          //any system sleep state.
    Method(_STA,0) {
        Return(Xor(GO01, One))       //Get power status
    }
    Method(_ON){                     //
        Store(Zero, GO01)            //Apply power
        Sleep(10)                    //
        Store(One, GO06)             //Pulse reset
        Stall(10)                    //
        Store(Zero, GO06)            //
    } // End Of _ON
    Method(_OFF){
        Store(One,GO01)              //Cut power
    } // End of _OFF
} // End of Power Resource PIDE
```

The Device object for IDE Drive 1 is named DRV1 in the following name space:

```
\_SB                        // System bus
    PCI0                    // PCI bus
        IDE1                // IDE PCI device
            _ADR                // Address of controller on the PCI bus (device #,function #)
            CHN0                // Channel
                _ADR            // 0 = primary channel
                DRV0            // IDE drive 0
                    _ADR        // 0 = Master
                DRV1            // IDE drive 1
                    _ADR        // 1 = Slave
```

To further illustrate the use of ACPI name space to model different configurations of IDE controllers and drives on a PCI bus, the following name space models an IDE controller with two channels. Two drives, a master and slave, are connected to each channel. Note that the Device objects for the drives (for example, _SB.PCI0.IDE1.CHN0.DRV1) are only needed in the name space if an ACPI control method is used to control the drive. Examples of such methods are the STA, ON, and OFF methods shown in the PowerResource object named PIDE in the previous sample of ASL code. A strong example of ASL code to handle an IDE can be found in the sample ASL code for the mobile concept machine (See the asl file called mb_smp.asl on the ACPI Website at http://www.teleport.com/~acpi/).

```
\_SB                        // System bus
    PCI0                    // PCI bus
        IDE1                // IDE PCI device
            _ADR                // Address of controller on the PCI bus (device #,function #)
            CHN0                // Channel
                _ADR            // 0 = primary channel
                DRV0            // IDE drive 0
                    _ADR        // 0 = Master
                DRV1            // IDE drive 1
                    _ADR        // 1 = Slave
            CHN1                // Channel
                _ADR            // 1 = secondary channel
                DRV0            // IDE drive 0
                    _ADR        // 0 = Master
                DRV1            // IDE drive 1
                    _ADR        // 1 = Slave
```

The following name space models two channels connected to an ISA bus. Two drives, a master and slave, are connected to each channel.

```
\_SB                            // System bus
    ISA                             // ISA bus
        CHN0                        // IDE ISA Primary Device
            _HID                    // _PNP0800
            _UID                    // 0 = Primary channel
            DRV0                    // Drive 0
                _ADR                // 0 = Master
            DRV1                    // Drive 1
                _ADR                // 1 = Slave
        CHN1                        // IDE ISA Secondary Device
            _HID                    // _PNP0800
            _UID                    // 1 = Secondary channel
            DRV0                    // Drive 0
                _ADR                // 0 = Master
            DRV1                    // Drive 1
                _ADR                // 1 = Slave
```

~~1.1.2~~8.3.2 Processor Power Management

Processor power management is accomplished by a combination of the Processor object in ACPI name space and the values in the DUTY_OFFSET and DUTY_WIDTH fields in the FACP table. For example, the ASL code that creates the processor object in the Trajan ACPI name space:

```
Scope(\_PR) {
    Processor(
      CPU0,          // processor name
      1,             // unique number for this processor
      0x1010,  // System IO address of Pblk Registers
      0x06           // length in bytes of PBlk
    ) {}
  }
```

Following are the values for the DUTY_OFFSET and DUTY_WIDTH fields in the Trajan ACPI table:

DUTY_OFFSET	1	1	The zero-based index of where the processor's duty cycle setting is within the processor's P_CNT register.
DUTY_WIDTH	1	3	The bit width of the processor's duty cycle setting value in the P_CNT register.

The value of 1 in the DUTY_OFFSET field of the FACP table indicates that the CLK_VAL value in the ACPI hardware Processor Control (P_CNT) register starts at bit 1. The value of 3 in the DUTY_WIDTH field indicates that the CLK_VAL field in P_CNT is 3 bits wide. In other words, bits 1, 2, and 3 of P_CNT are used for the CLK_VAL value.

~~1.1.3~~8.3.3 System Power Management

This section describes the role of the ACPI BIOS during transitions into and out of the system working state S0, and the system sleeping states S1, S2, S3, and S4.

~~1.1.1.1~~8.3.3.1 The BIOS's Role in Transitioning Out of the Working State (S0)

The OS has direct access to all the registers it needs to initiate a system sleep state. The OS always runs a _PTS control method in the ACPI name space at the beginning of each transition to a system sleep state.

An example of one thing that could be done in such a _PTS method in the Trajan ACPI BIOS would be to save SMM data changed since POST.

~~1.1.1.2~~8.3.3.2 The BIOS's Role in Waking from S1

The BIOS does not get control during the transition from S1 to S0.

Earlier, when the system switched from S0 to S1, no context was lost. So, at the start of the transition from S1 back to S0, the clocks are stopped and memory is in suspend refresh, but the chipset is on and the processor is on.

After the OS starts running in the S0 state, it will always run the _WAK control method in the ACPI name space (passing the value 1 as an argument). An example of what a Trajan BIOS _WAK method could do when invoked and passed a value of 1, is:

- Notify the OS to do a "device check" for changes in the docking station (by using the following form of the ASL **Notify** term: Notify(_SB.PCI0.MPC1, 1).
- Notify the OS to check and set thermal control (by using the following form of the ASL **Notify** term: Notify(_TZ.THRM, 0x81).

8.3.3.3 The BIOS's Role in Waking from S2

Earlier, when the system switched from S0 to S2, the processor was turned off (so all processor context is lost). At the start of the transition from S2 back to S0, the clocks are stopped and memory is in suspend refresh, the chipset is on, and the processor is off.

To start the transition from S2 to S0, the processor starts executing at the power-on reset vector, the processor SMRAM base register is restored, and control is passed to the OS. The OS:

- Gets the FACS table pointer from the CMOS.
- Retrieves the wake vector (a physical memory address) from the FACS table.
- Converts the physical wake vector address to a segment:offset format and starts executing at that address.

After the OS starts running in the S0 state, it will always run the _WAK control method in the ACPI name space (passing the value of 2 as an argument). An example of what a Trajan BIOS _WAK method could do when invoked and passed a value of 2, is:

- Notify the OS to do a "device check" for changes in the docking station (by using the following form of the ASL **Notify** term: Notify(_SB.PCI0.MPC1, 1).
- Notify the OS to check and set thermal control (by using the following form of the ASL **Notify** term: Notify(_TZ.THRM, 0x81).

8.3.3.4 The BIOS's Role in Waking from S3

Earlier, when the system switched from S0 to S3, the processor was turned off (so all processor context is lost) and the chipset was turned off (so all chipset register contents are lost). At the start of the transition from S3 back to S0, the clocks are stopped, and memory is in suspend refresh, the chipset is off, and the processor is off.

To start the transition from S3 to S0, the processor starts executing at the power-on reset vector, the memory controller configuration is restored, the cache is invalidated, the chipset register values are restored from the ACPI NVS memory area, the processor SMRAM base register is restored, and control is passed to the OS. The OS:

- Gets the FACS table pointer from the CMOS.
- Retrieves the wake vector (a physical memory address) from the FACS table.
- Converts the physical wake vector address to a segment:offset format and starts executing at that address.

After the OS starts running in the S0 state, it will always run the _WAK control method in the ACPI name space (passing the value of 3 as an argument). An example of what a Trajan BIOS _WAK method could do when invoked and passed a value of 3, is:

- Notify the OS to do a "device check" for changes in the docking station (by using the following form of the ASL **Notify** term: Notify(_SB.PCI0.MPC1, 1).

- Notify the OS to check and set thermal control (by using the following form of the ASL **Notify** term: Notify(_TZ.THRM, 0x81).

8.3.3.5 The BIOS's Role in Waking from S4

There are two methods for switching from S0 to S4:

- The BIOS saves memory context to disk, or
- The OS saves and restores memory.

An ACPI-compatible platform specifies the alternative it uses by setting a value in the S4BIOS_REQ field of the FACP table. In this case study of the Trajan platform, the S4BIOS_REQ field is set to zero, so the OS saves and restores memory (for FACP table field values used by this example Trajan platform, see the topic "Sizing Memory, Allocating Memory, Fixing Up Table Pointers, and Copying ACPI Tables into Memory").

For the BIOS, using the OS method to switch from S4 to S0 is the same process as a cold boot (for a description of the cold boot process, see the topic "Initializing the ACPI BIOS During POST and Cold Boot Sequence").

Note that in waking from S4, the OS makes use of the "hardware signature" value calculated by the BIOS at the previous boot and stored in the FACP table. The definition of the Hardware Signature field in the FACP table is reprinted below from the *ACPI Specification, Revision 1.0*.

> The value of the system's "hardware signature" at last boot. This value is calculated by the BIOS on a best effort basis to indicate the base hardware configuration of the system such that different base hardware configurations can have different hardware signature values. The OS uses this information in waking from an S4 state, by comparing the current hardware signature to the signature values saved in the non-volatile sleep image. If the values are not the same, the OS assumes that the saved non-volatile image is from a different hardware configuration and can not be restored.

The OS compares the current hardware signature value to the hardware signature value stored in the FACS at last boot.

- If the two values are the same, the OS restores the contents of conventional memory from the ACPI NVS memory area.
- If the two values are not the same, the OS does not restore any memory and cold boots the system.

In either case, once the OS starts running in the S0 state, it will always run the _WAK control method in the ACPI name space (passing the value of 4 as an argument).

There are other checking methods that can be used in addition to comparing the hardware signature value ….

~~1.4~~8.4 Plug and Play Using ACPI on the Trajan Motherboard

To enable the OS to successfully carry out device configuration on motherboard devices, an ACPI Device object *must* be defined in the ACPI name space for each device on the motherboard. This section describes the device ID and configuration objects that must be defined under the Device object for

- Single-configuration objects.
- Multiple-configuration objects.

8.4.1 Name Space Objects for Single-Configuration Devices

On the Trajan motherboard, the speaker device on the ISA bus is an example of a device with a single configuration. For such a device, three objects must be declared under the Device object in the name space:

- _HID, which reports the device Plug and Play ID
- _CRS, which reports the device's single configuration.
- _STA, which reports the device's status: enabled, disabled, and so on.

The following sample ASL code illustrates this:

```
Device(SPKR) {                                  // Device object name is 'SPKR'
    Name(_HID,EISAID("PNP0800"))                // Hardware Device ID for speaker class
    Method(_STA, 0) {
        Return(0x0F)                            // Present, enabled, functioning
    }
    Name(_CRS,Buffer(){
        0x47,           // IO port type of resource descriptor
        0x01,           // Decodes 16-bit addresses
        0x61,           // Bits [7:0] of minimum base I/O address SPKR can be configured for
        0x00,           // Bits [15:8] of minimum base I/O address
        0x61,           // Bits [7:0] of maximum base I/O address SPKR can be configured for
        0x00,           // Bits [15:8] of maximum base I/O address
        0x01,           // Base alignment
        0x01,           // Number of contiguous I/O ports requested
        0x79,           // End tag
        0x00
        } // end of Buffer
    ) // end of _CRS
} // end of SPKR device
```

~~1.1.2~~8.4.2 Name Space Objects for Multiple-Configuration Devices

On the Trajan motherboard, the floppy disk controller in the Super IO chip is an example of a device with multiple configurations. For such a device, these objects and control methods need to be implemented under the Device object in the name space:

- _HID, which reports the device Plug and Play ID
- _CRS, which reports the device's current configuration.
- _PRS, which reports the device's possible configurations.
- _SRS, which sets the device's current configuration.
- _STA, which reports the device's current status (enabled and decoding hardware resources or disabled and not decoding hardware resources, functioning properly or failed its diagnostics, and so on).
- _DIS, which disables the device (called after the OS sets the device to a D3 device power state).

8.4.3 Field Declarations

ASL **OperationRegion** and **Field** terms (not shown) declare the following named fields for use by the _CRS, _SRS, _STA, and _DIS control methods that go under the Device object for the floppy disk controller in the name space hierarchy:

- 'FER' is a SuperIO chip register that contains the FDC enable bit.
- 'F0AD' is a field in the chipset register RESB that selects the FDC IO address.
- 'F0EN' is a field in the chipset register RESB that enables the decoding of the selected IO address.

8.4.4 Example _CRS, _SRS, _STA, and _DIS Methods for the FDC

The following sample ASL code illustrates this for the floppy disk controller on the example Trajan motherboard:

```
Device(FDC0) {                                  // Floppy Disk Controller (FDC)
    Name(_HID, EISAID("PNP0700"))               // PnP Device ID

    Method(_CRS,0){                             //Current Resource Setting for FDC
        Name(BUF0,Buffer(){
            //IO port descriptor, 16-bit decode, IO port 370-377
            0x47, 0x01, 0x70, 0x03, 0x70, 0x03, 0x01, 0x08,
            //IRQ descriptor, edge, IRQ6
            0x22, 0x40, 0x00,
            //DMA descriptor, channel 2, ISA compatible
            0x2A, 0x04, 0x00,
            //End tag
            0x79, 0x00}
        )
        CreateByteField(BUF0, 0x02, IOLL)   //Low byte of min port
        CreateByteField(BUF0, 0x04, IOHL)   //Low byte of max port
        And(FER, 0x20, local0)                  //Check FDC select bit
        If(LEqual(Local0,Zero)) {           //Primary FDC selected
            Store(0xF0,IOLL)                    //Change port address
            Store(0xF0, IOHL)
            }
        Return(BUF0)
    } //End _CRS method
        .
        .
        .
    Method(_SRS,1){                                 //Set Resource
        //Arg0 contains PnP resource string passed in from the OS
        CreateWordField(Arg0, 0x02, IOAR)       //IOAR points to port
        And(FER, 0xD7, Local0)                      //Assume primary FDC
        Store(Zero, Local1)
        If(LEqual(IOAR, 0x370)){                //If OS selects secondary FDC
            Or(Local0, 0x20, Local0)            //Change for secondary FDC
            Store(One, Local1)
        Store(Local1, PCI0.PX43.RESB.F0AD)      //Set FDC address
        Store(One, PCI0.PX43.RESB.F0EN)         //Enable FDC decode
        Or(Local0, 0x08, Local0)                //Enable FDC
        Store(Local0, FER)
        Store(Local0, DAT)
    } // end of _SRS method

    Method(_STA,0){                             // Status of the FDC
        And(FER, 0x08, Local0)                  // Check FDC enable bit
        If(Local0)
            {Return(0x0B)}                      // Present, enabled, and functioning
        Else
            {Return(0x01)}                      // Present, but disabled
    } //end _STA method

    Method(_DIS,0){                             // Disable FDC
        And(FER, 0xF7, Local0)
        Store(Local0, FER)
        Store(Local0, DAT)                      // Need two writes to data port!
        Store(Zero, PCI0.PX43.RESB.F0EN)    // Disable decode
    } //end _DIS method
} // end of FDC0 device
```

~~1.5~~8.5 Docking Using ACPI on the Trajan Motherboard

To enable the OS to successfully manage devices inserted into and removed from a docking station, an ACPI Device object must be declared for the Dock in the ACPI name space. Under the Device object representing the Dock,

- Device configuration objects must be used.
- Device insertion and removal objects must be used (for example, _EJx).
- The _UID device identification method may need to be used to report the unique dock serial number.

The interface with the OS is accomplished using ASL synchronization and notification terms.

This section illustrates these ideas with example ASL code for the Trajan example platform.

~~1.1.1~~8.5.1 Field Declarations

ASL **OperationRegion** and **Field** terms (not shown) declare the following named fields:

- 'OPEN' is a field in the docking chipset registers (OPENREQ.P41.0) that is an undock request.
- 'DOCK' is a field in the docking chipset registers (DOCKED.P41.1) that reports system docked.
- 'UDKP' is a field in the docking chipset registers (UDKPERMIT.P41.3) that reports whether an undock request is permitted.
- 'QENx' names a set of fields in the docking chipset registers (QvccENx.P40.[2-3]) that are Q buffer enables.
- 'MIEN' is a field that reports whether a PCI-ISA bridge is present.

~~1.1.2~~8.5.2 Example _ADR, _UID, _EJ0, and Device Objects for the Dock

```
Device(MPCI) {                          //Docking station
    Name(_ADR, 0x00110000)              //Bridge is Dev 17 on PCI bus 0
    Name(_UID, 1)                       //Unique ID is 1
    Method(_EJ0, 1){                    //Hot docking support
        //Arg0: 0=insert, 1=eject
        If(Arg0) {                      //Eject
            Store(One, UDKP)            //Start undock sequence
            Wait(EJT0, 0xFFFF)          //Wait for signal from OS
            Return(0)
            }
        Else{}}                                 //Insert, nothing to do
    //Declare Device objects for all devices behind docking here
    Device(MISA) {
        .
        .
        .
    } //End device MISA
    Device(DIDE) {
        .
        .
        .
    } //End device DIDE
        .
        .
        .
} //End device MPCI
```

~~1.1.3~~8.5.3 Example Synchronization and Notifications

```
Event(EJT0)                                     //Declare synchronization object
Scope(_GPE) {
    Method(_L09) {
    // Get docking status
    XOr(OPEN, 0x01, Local1)         //Undock request inverted
    XOr(DOCK, 0x01, Local2)         //System docked report inverted
    XOr(UDKP, 0x01, Local3)         //Undock request permitted report inverted
    If(Local1) {                            //This is an undock request
        Store(Zero, OPEN)                   //Clear status bit
        Notify(\_SB.PCI0.MPCI, 1) //Notify OS of event on MPCI (Dock) device object
                                            //1 = Ejection Request
        }
    Else {If(Local2) {                      //This is a station docked event
        Store(One, QEN1)                    //Turn on Q buffers
        Store(One, QEN2)
        Store(One, MIEN)                    //Enable ISA bus
        Notify(\_SB.PCI0.MPCI, 0) //Notify OS of event on MPCI (Dock) device object
                                            //0 = Device Check
        }
    Else {If(Local3) {                      //Start undock sequence
        Store(Zero, QEN1)                   //Turn off Q buffers
        Store(Zero, QEN2)
        Store(Zero, UDKP)                   //Undock sequence starts here
        Stall(200)                          //Give chipset time to finish,
        Stall(100)                          //No blocking
        Signal(EJT0)                        //Release synchronization object
        }}
    }
}
```

~~1.6~~8.6 Switching Between ACPI and Legacy Modes on the Trajan Motherboard

The OS switches a platform between ACPI and legacy modes by writing values to the SMI command (SMI_CMD) port. The system port address of the SMI command port for a particular platform is in the SMI_CMD field of the FACP table (for example, the SMI_CMD system port address for this example Trajan

platform is 0xB2; see the topic “Sizing Memory, Allocating Memory, Fixing Up Table Pointers, and Copying ACPI Tables into Memory”).

- On platforms that support both ACPI and legacy modes, as well as on platforms that support ACPI-only, the OS writes the value ACPI_ENABLE to the SMI_CMD port address to switch to ACPI mode.
- On platforms that support both ACPI and legacy modes, the OS writes ACPI_DISABLE to the SMI_CMD port address to switch to legacy mode.

8.6.1 Switching From Legacy to ACPI Mode

When switching from legacy to ACPI mode, the BIOS

1. Saves all power management-related chipset registers (such as idle, traps, timers, and so on) in SMRAM.
2. Disables all legacy PM timers.
3. Disables all ACPI events so the OS can select the events that match its policies.
4. Sets the SCI_EN bit in the ACPI hardware register.

8.6.2 Switching From ACPI to Legacy Mode

When switching from ACPI to legacy mode, the BIOS

- Restores the power-management related registers from SMRAM; this puts idles, traps, timers, and so on back to their original states.
- Disables the SCI_EN bit in the ACPI hardware register.

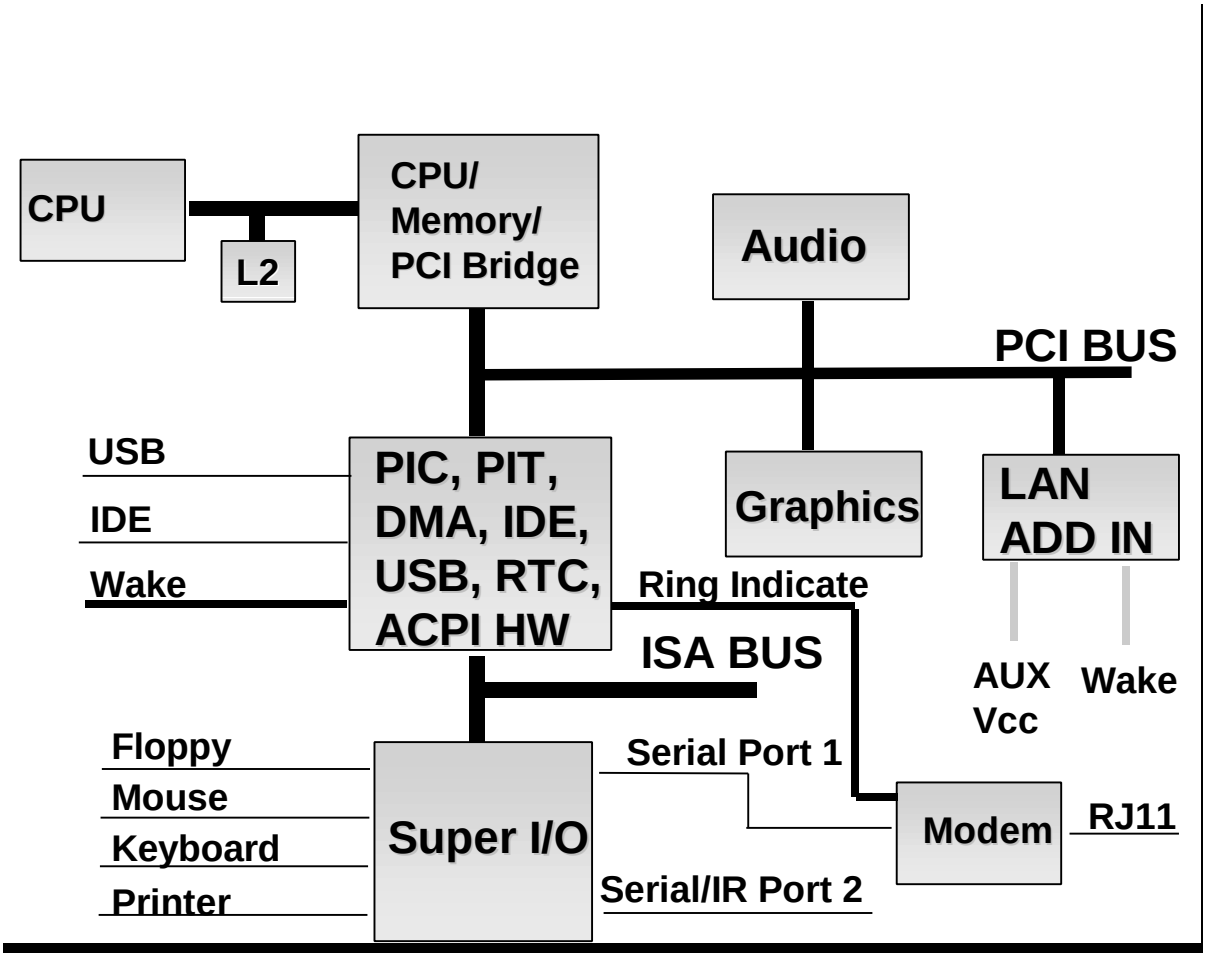
.

9. ACPI Desktop Concept Machine

This section presents the ACPI desktop concept machine. The desktop concept machine can be characterized as follows:

- Single processor.
- Single root bridge.
- ACPI hardware support built into the chipset.
- Wake events can come from the LAN controller or the modem.
- Implements the following power saving states:
- System states S1, S4, and S5.
- Processor states C0, C1, and C2.
- Device states D0 and D3.

The relationships between the Desktop concept machine components are illustrated in the following simplified block diagram:



~~1.1~~9.1 Desktop Concept Machine Design Overview

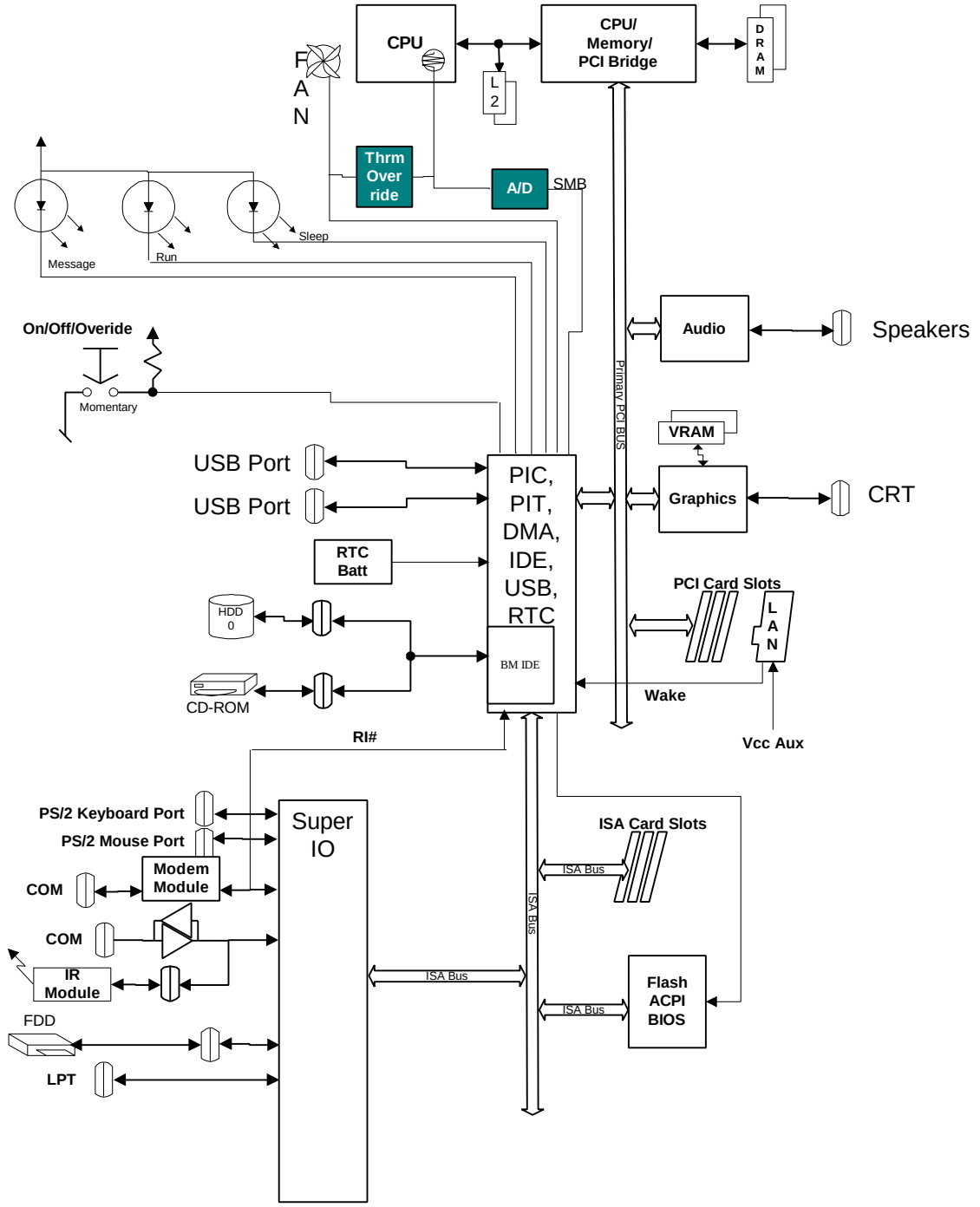
This section describes the hardware components used on the Desktop concept machine and shows the relationships between components with a relatively detailed block diagram.

~~1.1.1~~9.1.1 Hardware Devices

The prominent hardware devices that characterize the desktop concept machine are listed in the following table.

Device	Description
Chipset	Intel
Super I/O	SMC FDC37C93XFR
Video	C&T 65548 Flat Panel/CRT GUI Accelerator
LAN	Assumed a standard LAN card with a "Magic Packet" output. Magic packet output routed to chip set to generate wake event.
IR	Generic interface shared with one of the COM ports.
Modem	Standard modem chip set interfaced through one of the serial ports. Ring Indicate wired direct to RI# wake up input on the chip set.
A/D	National LM75 Temperature sensor output connected through SMBus.
USB	Controller with two ports built into chip set.
Audio	PCI device with no added power resources

~~1.1.2~~9.1.2 Desktop Block Diagram



~~1.2~~9.2 Desktop Concept Machine ACPI Name Space

This section shows the ACPI name space that models the desktop concept machine hardware components and their relationships.

~~1.1.1~~9.2.1 Structure of the Data and Address Buses in ACPI Name Space

The backbone of ACPI name space is the hierarchy of device objects for the data and address buses, which is shown below. The desktop concept machine is a single-processor, single-root bridge machine and that shows in the diagram below.

```
\_PR                            //Single processor object
    .
    .
    .
\_SB
    PCI0                        //Root PCI bus object
        .
        .
        .
        ISA                         //PCI-ISA bridge object
            .
            .
            .
        USB0                        //USB Device object
        LAN0                        //LAN device object
```

~~1.1.2~~9.2.2 All the Objects in the ACPI Name Space

This section shows all the objects in the hierarchical ACPI name space for the desktop concept machine.

```
\_PR
\_S0
\_S1
\_S2
\_S4
\_S5
\_PTS
\_WAK
TP1H
TP1L
TP2H
TP2L
TPC
TVAR
\_TZ                    //Thermal Zone
    TSAD
    SBYT
    WBYT
    RBYT
    RWRD
    SCFG
    STOS
    STHY
    RTMP
    PFAN                //Power Resource for processor fan
        _STA            //Status
        _ON             //Fan On
        _OFF            //Fan Off
    FAN0
        _HID            //Hardware Device ID
        _PR0            //Reference to power resource for D0
    THM1
        _AL0            //Active cooling device list  (e.g. FAN)
        _AC0            //
        _PSL
        _TSP
        _TC1
        _TC2
        _PSV
        _CRT            //Critical Temp.
        _TMP            //Get Current temp
        _SCP
        STMP
\_SI                    //System Indicator
    _MSG
    _SST
\_GPE
    _L00
\_SB
    LNKA
    LNKB
    LNKC
    LNKD
    PCI0
        _HID
        _ADR            //Device address on the PCI bus
        _CRS            //Current Resource (Bus number 0)
        _PRT
        PX40
            _ADR
        USB0
            _ADR
            _STA
            _PRW
        PX43
            _ADR
        ISA
            PIC
                _HID
                _CRS
            MEM
```

```
        _HID
        _CRS
    DMA
        _HID
        _CRS
    TMR
        _HID
        _CRS
    RTC
        _HID
        _CRS
    SPKR
        _HID
        _CRS
    COPR
        _HID
        _CRS
    ENFG
    EXFG
    JOY1
    FDC0                //Floppy Disk controller
        _HID            //Hardware Device ID
        _STA            //Status of the Floppy disk controller
        _DIS        //Disable
        _CRS            //Current Resource
        _PRS            //Possible Resource
        _SRS            //Set Resource
    UAR1                //Communication Device (Modem Port)
        _HID            //Hardware Device ID
        _STA            //Status of the Communication Device
        _DIS        //Disable
        _CRS            //Current Resource
        _PRS            //Possible Resource
        _SRS            //Set Resource
        _PRW            //Wake-up control method
    IRDA                //IRDA device
        _HID            //Hardware Device ID
        _STA            //Status of the COM device
        _DIS        //Disable
        _CRS            //Current Resource
        _PRS            //Possible Resource
        _SRS            //Set Resource
    COMB
        _HID            //Hardware Device ID
        _STA            //Status of the COM device
        _DIS        //Disable
        _CRS            //Current Resource
        _PRS            //Possible Resource
        _SRS            //Set Resource
    FIRA                //Fast IR device
        _HID            //Hardware Device ID
        _STA            //Status of the COM device
        _DIS        //Disable
        _CRS            //Current Resource
        _PRS            //Possible Resource
        _SRS            //Set Resource
    LPT                 //Standard Printer
        _HID            //Hardware Device ID
        _STA            //Status of the printer device
        _DIS        //Disable
        _CRS            //Current Resource
        _PRS            //Possible Resource
        _SRS            //Set Resource
    ECP                 //Extended capabilities Port
        _HID            //Hardware Device ID
        _STA            //Status of the port device
        _DIS        //Disable
        _CRS            //Current Resource
        _PRS            //Possible Resource
        _SRS            //Set Resource
    PS2M                //PS2 Mouse Device
```

```
                    _HID
                    _STA
                    _CRS
                PS2K
                    _HID
                    _CRS
                    _STA
// IDE, Video and Audio don't appear as there is no value added hardware and
// system uses the standard PCI PnP and standard driver support power
// management
            LAN0
                _ADR
                _PRW
```

~~1.3~~9.3 Implementation Examples from the Desktop Concept Machine

This section uses blocks of example ASL code, name space diagrams, and hardware component block diagrams to discuss in some detail the following aspects of the Desktop concept machine:

- Power resource implementation.
- Thermal zone implementation.
- Power Button support.
- Operation region and field definitions for a Super I/O chip.

9.3.1 Power Resource Implementation

The ACPI specification defines a power resource as system components (for example, power planes and clock sources) that a device requires to operate in a given power state. An example of a device on the Desktop concept machine that operates in different power states is system fan (the device named "FAN0" in the ACPI name space). FAN0 has two power states: D0 (fully on) and D3 (fully off). The power state of the fan is switched by writing a 0 or 1 to bit 0 of the GPOB. Following are the lines of sample ASL code that work together to control the system fan (the line numbers are artifacts to make it easier to walk through the code):

- Lines 3 through 6 declare a field in System I/O space named 'FANM' that can be read to check fan power status and written to switch the fan device power states.
- Lines 8 through 19 declare the PowerResource object named 'PFAN' for the fan device (declared as 'FAN0' in lines 20 through 23). Typically, a PowerResource object contains methods for device power (D*x*) states and for determining the current power state of the device. Notice that in the 'PFAN' PowerResource object, the 'FANM' field is used to switch the fan between D0 (fully on) and D3 (fully off) and to determine the status.
- Line 22 links the device named 'FAN0' to the PowerResource named 'PFAN'.

```
1       Device(PX43) {                                          // Map f3 space
2           Name(_ADR, 0x00070003)
                .
                .
                .
3           OperationRegion(GPOB,SystemIO,0x00,4)       //Operation Region for
4           Field(GPOB,ByteAcc,NoLock,Preserve){        //Fields for GP output bits
5               //bit assignments are here are based on system wiring
6               FANM,1,             //Fan Motor control
                    .
                    .
                    .
7       }               // End of PX43
            .
            .
            .
8       PowerResource(PFAN, 0, 0){          //Power Resource for Processor Fan
9           Method(_STA,0) {
10              Return(FANM)                    // get fan status
11          }
12          Method(_ON){                        // Switch on the FAN
13              Store(1,FANM)                   // Bit 0 of GPOB is used to control the
14                                                  // Fan Motor
15          } // End Of _ON
16          Method(_OFF){
17              Store(0,FANM)                   // reset bit0, turn off FAN
18          } // End of _OFF
19      } // End of Power Resource  PFAN
20      Device(FAN0) {
21          Name(_HID, "PNP0C0B")
22          Name(_PR0, Package(1){PFAN})
23      }
```

~~1.1.2~~9.3.2 Thermal Zone Implementation

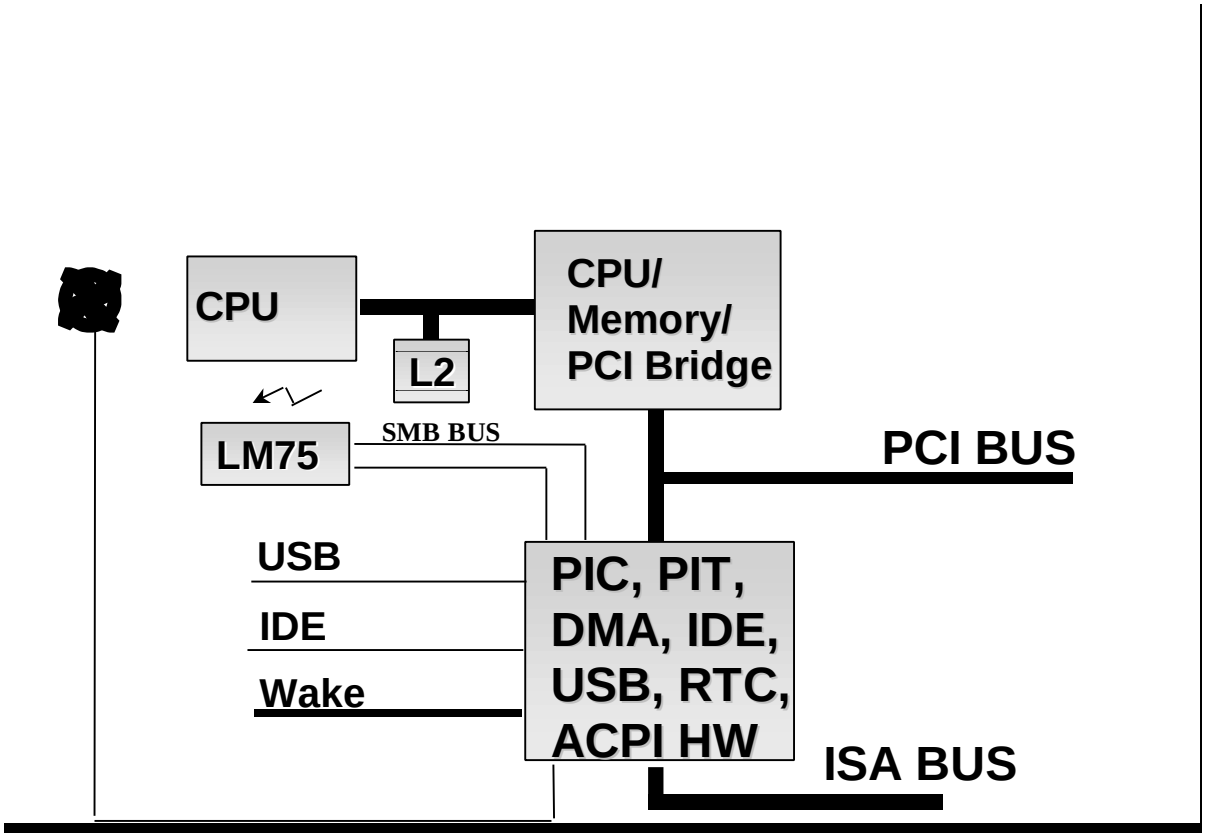
This section describes the thermal zone implementation on the desktop concept machine.

~~1.1.1.1~~9.3.2.1 Physical Components of Thermal Management

Physically, on the desktop concept machine, thermal management is accomplished by using an LM75 as a temperature sensor.

- The LM75 communicates with the system over the SMBus.
- General purpose output signals are used to drive the cooling fan.

The relationships between these physical components is shown in the following block diagram.



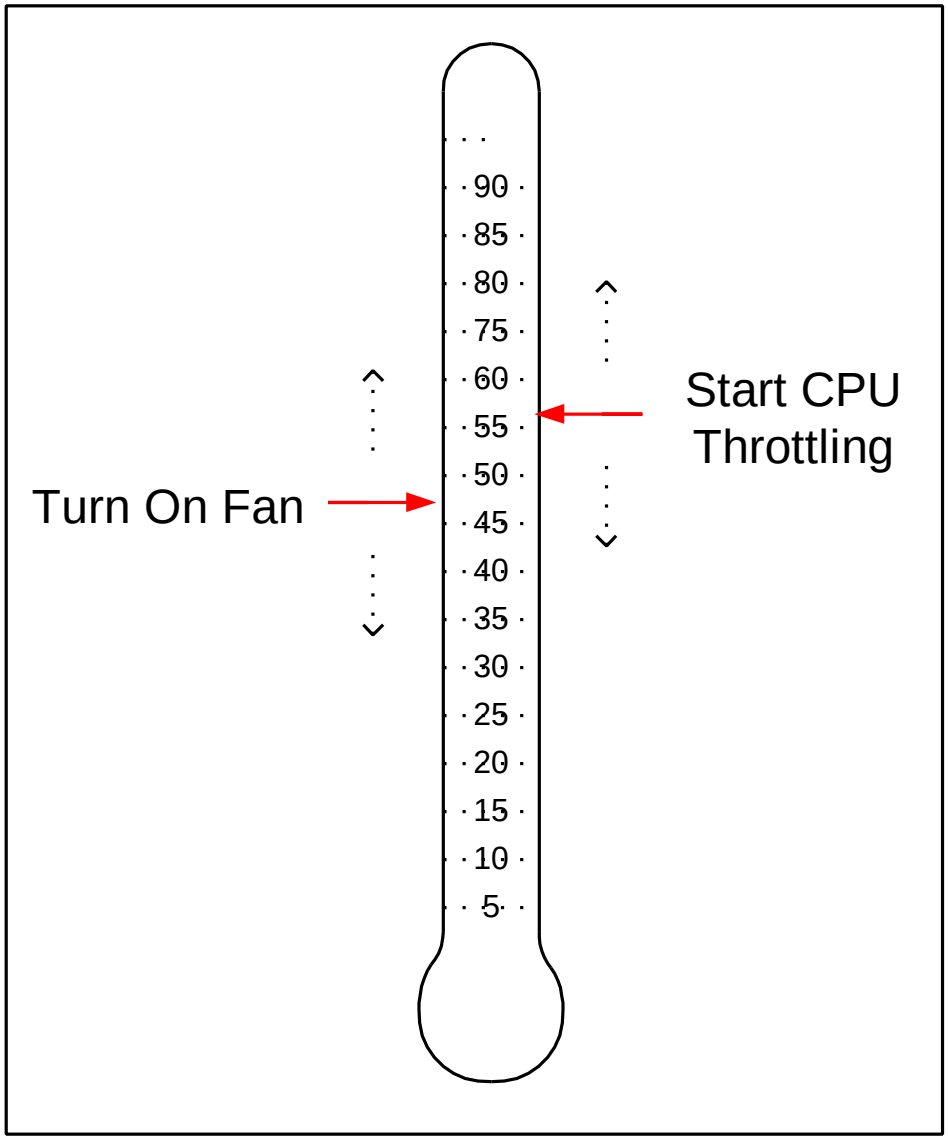
~~1.1.1.2~~9.3.2.2 Defining a Thermal Policy for the Desktop Concept Machine

The desktop concept machine thermal policy is based on the temperature "trip point" model shown in the following illustration. The components of the temperature policy are:

- A temperature value ("trip point") at which to start the fan (that is, start "active cooling").
- A temperature value ("trip point") at which to start throttling the processor (that is, start "passive cooling").
- The ability to change either of these values at any time.

The illustration shows the active trip point set at about 47 degrees C and the passive trip point set at about 57 degrees C. The dotted arrows indicate that these trip point values can slide up and down the temperature scale over time (in the desktop concept machine implementation, changes to these values is accomplished by ASL control methods).

The LM75 temperature sensor device on the desktop concept machine has two registers in which trip point temperature values can be stored and the sensor generates an event whenever the temperature of the thermal zone it is monitoring rises above or falls below one of these values.



~~1.1.1.3~~9.3.2.3 Writing ASL Code that Carries Out the Thermal Policy

The objects that define the thermal policy for a platform are placed under the _TZ scope in the ACPI name space hierarchy.

```
TP1H                    //Trip point 1 high value
TP1L                    //Trip point 1 low value
TP2H                    //Trip point 2 high value
TP2L                    //Trip point 2 low value
TPC                     //Critical trip point
TVAR                    //Thermal variables and policy flag
\_TZ                  //Thermal Zone
    TSAD
    SBYT
    WBYT
    RBYT
    RWRD
    SCFG
    STOS
    STHY
    RTMP
    PFAN                //Power Resource for processor fan
        _STA            //Status
        _ON             //Fan On
        _OFF            //Fan Off
    FAN0
        _HID            //Hardware Device ID
        _PR0            //Reference to power resource for D0
    THM1
        _AL0            //Active cooling device list  (e.g. FAN)
        _AC0            //Returns active trip point
        _PSL          //Passive cooling device list (e.g. PR0)
        _TSP            //Passive policy sampling period
        _TC1            //Passive cooling thermal constant
        _TC2            //Passive cooling thermal constant
        _PSV            //Returns passive trip point
        _CRT            //Returns critical trip point
        _TMP            //Returns current temperature
        _SCP            //Sets user cooling policy to active or passive
        STMP            //Sets trip point value into High or Low register in LM75
                        //Uses SMBus interface to communicate with LM75
```

It takes a relatively large number of objects in the ACPI name space to fully implement a thermal policy, even one as straight-forward as the one for the desktop concept machine. One of the traps first-time ACPI developers can fall into is to not use all the objects in the _TZ scope that are required to carry out their thermal policy. Following is a list of built-in objects to use within the _TZ scope:

Object	Description
_AC*x*	Returns Active trip point in tenths Kelvin
_AL*x*	List of pointers to active cooling device objects
_CRT	Returns critical trip point in tenths Kelvin
_PSL	List of pointers to passive cooling device objects
_PSV	Returns Passive trip point in tenths Kelvin
_SCP	Sets user cooling policy (Active or Passive)
_TC1	Thermal constant for Passive cooling
_TC2	Thermal constant for Passive cooling
_TMP	Returns current temperature in tenths Kelvin
_TSP	Thermal sampling period for Passive cooling in tenths of seconds

Following is the ASL code that maintains the state of the Thermal Zone. The buffer TVAR has three fields: PLCY (set the 0 if the current user policy is Active and set to 1 if the current user policy is Passive), and CTOS and CTHY (which hold the values for the two LM75 registers). Note the ASL coding technique of the ASL **CreateByteField** and CreateWordField terms to create named fields (substrings) out of a string in a Buffer.

```
//
// Thermal Zones
// Define thermal constants, flag and variables
Name(TP1H, 3332)                    // Trip point 1 high = 60.0c
Name(TP1L, 3282)                    // Trip point 1 low = 55.0c
Name(TP2H, 3432)                    // Trip point 2 high = 70.0c
Name(TP2L, 3382)                    // Trip point 2 low = 65.0c
Name(TPC, 3532)                     // Critical trip point = 80.0c
Name(TVAR, Buffer(){1, 32, 33, 82, 32}) // Thermal variables and flag
CreateByteField(TVAR, 0, PLCY)          // Default policy to passive
CreateWordField(TVAR, 1, CTOS)          // Current Tos = 40.0c
CreateWordField(TVAR, 3, CTHY)          // Current Thyst = 35.0c
```

The ASL code that implements the thermal policy for the Desktop concept machine is shown below (line numbers are artifacts to make it easier to refer to particular lines of code).

Lines 56 through 64 is a control method that sets one of the trip point registers in the LM75. This method, named ‘STMP’, takes two arguments: which trip point register to set in the LM75 (high or low); the temperature value to put into the register. The STMP method uses the STOS or STHY method to move the new value into the appropriate LM75 register (the ASL code for the STOS and STHY methods is shown in a later section; an SMBus interface is used by these methods to communicate with the LM75 and that is beyond the scope of this discussion).

```
1   Scope(\_TZ)
2   {
3   // Start of _TZ
4       PowerResource(PFAN, 0, 0){          //Power Resource for Processor Fan
5           Method(_STA,0) {
6               Return(FANM)                    // get fan status
7           }
8           Method(_ON){                        // Switch on the FAN
9               Store(1,FANM)                   // Bit 0 of GPOB is used to control the
10                                                  // Fan Motor
11          } // End Of _ON
12          Method(_OFF){
13              Store(0,FANM)                   // reset bit0, turn off FAN
14          } // End of _OFF
15      } // End of Power Resource  PFAN
16      Device(FAN0) {
17          Name(_HID, "PNP0C0B")
18          Name(_PR0, Package(1){PFAN})
19      }
20      ThermalZone(THM1) {
21          // Kelvin = Celsius + 273.2
22          // Active cooling objects _AL0 and _AC0
23          Name(_AL0, Package(1){FAN})     //Active cooling device list
24          Method(_AC0, 0) {                       //Returns active trip point
25              If(Or(PLCY, PLCY, Local7)) {    //Passive policy is current
26                  Return(TRP1)}
27              Else {                                  //Active policy is current
28                  Return(TRP2)}
29          } // Method(_AC0)
30          // Passive cooling objects
31          Name(_PSL, Package(1){\_PR.CPU0})//Passive cooling device list
32          Name(_TSP, 30)                          //Returns passive cooling sampling period
33          Name(_TC1, 4)                       //Returns passive cooling thermal constant
34          Name(_TC2, 4)                       //Returns passive cooling thermal constant
35          Method(_PSV, 0) {                   //Returns passive trip point
36              If(Or(PLCY, PLCY, Local7)) {    //Passive policy is current
37                  Return(TRP2)}
38              Else {                                  //Active policy is current
39                  Return(TRP1)}
40          } // Method(_PSV)
41          Method(_CRT, 0) {                   //Returns critical trip point
42              Return(TRPC)
43          }
44          Method(_TMP, 0) {
45              Return(RTMP())
46          } // Method(_TMP)                   //Returns current temperature
47          Method(_SCP, 1) {                   //Sets current user policy
48              If(Arg0) {                      //Current policy is passive
49                  Store(One, PLCY)}
50              Else {                                  //Current policy is active
51                  Store(Zero, PLCY)
52              }
53              Notify(\_TZ.THRM, 0x81)     // Notify trip point change
54          } // Method(_SCP)
55          // Set temperature trip point
56          Method(STMP, 2) {
57              // Arg0 = trip point type (0 - high, 1 - low)
58              // Arg1 = temperature value word
59              Store(Arg1, DW00)
60              If(Arg0) {      // Set trip point low
61                  STHY(DB00, DB01)}
62              Else {          // Set trip point high
63                  STOS(DB00, DB01)}
64          } // Method(STMP)
65      } // ThermalZone(THRM)
66  } // Scope(\_TZ)
```

The ASL code in the following two Include files provides the communication with the LM75 over the SMBus.

~~1.1.1.1.1~~9.3.2.3.1 LM75 Thermal Device Include File

This section shows the contents of the Include file Lm75.asl.

```
// SMB support for thermal sensor

// Thermal device SMB device address = 90h
Name(TSAD, 0x90)

// SMB bus protocol
Include("px4smb.asl")

// Set register pointer
//Method(SRPR, 1) {
//      // Arg0 = index (0 - 3) into register block
//      Store(Arg0, Local1)             // Command byte
//
//      Or(TSAD, 1, Local0)             // Address byte + write command
//      SBYT(Local0, Local1)                    // Start "send byte" protocol
//}
//
// Set configuration register
Method(SCFG, 1) {
        // Arg0 = configuration byte
        WBYT(TSAD, 0x01, Arg0)
}

// Set Tos register
Method(STOS, 2) {
        // Arg0 = temperature low byte
        // Arg1 = temperature high byte
        // Set pointer register (occupy the command byte) to 0x03
        // Somehow LM75 wants to send out the MSB byte first !
        WWRD(TSAD, 0x03, Arg1, Arg0)
}

// Set Thyst register
Method(STHY, 2) {
        // Arg0 = temperature low byte
        // Arg1 = temperature high byte
        // Set pointer register (occupy the command byte) to 0x02
        // Somehow LM75 wants to send out the MSB byte first !
        WWRD(TSAD, 0x02, Arg1, Arg0)
}

// Read temperature register
Method(RTMP, 0) {
        // Set pointer register (occupy the command byte) to 0x00
        // DATW, DB00 and DB01 are defined as global buffer fields
        Store(RWRD(TSAD, 0x00), DW00)

        // Somehow LM75 returns right byte first! Need to swap bytes
        Store(DB00, Local0)
        Store(DB01, Local1)
        Store(Local1, DB00)
        Store(Local0, DB01)

        // After shift, DW00 has temperature*2 in Celsius
        // Bit 8-1 = whole value, bit 0 = decimal value in 0.5c
        // For example, 000110011b (51 decimal) => 25.5c
        ShiftRight(DW00, 7, DW00)

        // Multiply temperature by 10 to
        // convert it to format xx.y (255 => 25.5c)
        // After shift, DW01 has temperature*8 in Celsius
        ShiftLeft(DW00, 2, DW01)
        Add(DW01, DW00, DW00)

        // Convert to Kelvin in format xxx.y (2732 => 273.2k)
        Add(DW00, 2732, DW00)

        Return(DW00)
} // Method(RTMP)
```

~~1.1.1.1.29~~.3.2.3.2 SMBus Protocol Include File

This section shows the contents of the Include file Px4smb.asl.

```
// PIIX4 SMB interface methods

// Send byte protocol
Method(SBYT, 2) {
        // Arg0 = address byte
        // Arg1 = command byte
        Store(Arg0, SM04)           // Device address
        Store(Arg1, SM03)           // Command byte

        Store(0xFF, SM00)           // Clear all status bits
        Store(0x44, SM02)           // Byte command + start

      And(SM00, 0x02, Local0)       // Wait till completion
      While(LEqual(Local0, Zero)) {
            Stall(1)
            And(SM00, 0x02, Local0)}
} // Method(SBYT)

// Write byte protocol
Method(WBYT, 3) {
        // Arg0 = address byte
        // Arg1 = command byte
        // Arg2 = data byte
        Store(Arg0, SM04)           // Device address
        Store(Arg1, SM03)           // Command byte
        Store(Arg2, SM05)           // Data byte

        Store(0xFF, SM00)           // Clear all status bits
        Store(0x48, SM02)           // Byte data command + start

      And(SM00, 0x02, Local0)       // Wait till completion
      While(LEqual(Local0, Zero)) {
            Stall(1)
            And(SM00, 0x02, Local0)}
} // Method(WBYT)

// Write word protocol
Method(WWRD, 4) {
        // Arg0 = address byte
        // Arg1 = command byte
        // Arg2 = data low byte
        // Arg3 = data high byte
        Store(Arg0, SM04)           // Device address
        Store(Arg1, SM03)           // Command byte
        Store(Arg2, SM05)           // Data low byte
        Store(Arg3, SM06)           // Data high byte

        Store(0xFF, SM00)           // Clear all status bits
        Store(0x4C, SM02)           // Word data command + start

      And(SM00, 0x02, Local0)       // Wait till completion
      While(LEqual(Local0, Zero)) {
            Stall(1)
            And(SM00, 0x02, Local0)}
} // Method(WWRD)

// Read byte protocol
Method(RBYT, 2) {
        // Arg0 = address byte
        // Arg1 = command byte
        Or(Arg0, 0x01, Local1)          // Read command
        Store(Local1, SM04)         // Device address
        Store(Arg1, SM03)           // Command byte

        Store(0xFF, SM00)           // Clear all status bits
        Store(0x48, SM02)           // Byte data command + start

      And(SM00, 0x02, Local0)       // Wait till completion
      While(LEqual(Local0, Zero)) {
            Stall(1)
            And(SM00, 0x02, Local0)}
```

```
        Return(SM05)                // Return data in DAT0
} // Method(RBYT)

// Read word protocol
Method(RWRD, 2) {
        // Arg0 = address byte
        // Arg1 = command byte
        Or(Arg0, 0x01, SM04)            // Device address + Read command
        Store(Arg1, SM03)           // Command byte

        Store(0xFF, SM00)           // Clear all status bits
        Store(0x4C, SM02)           // Word data command + start

       And(SM00, 0x02, Local0)      // Wait till completion
       While(LEqual(Local0, Zero)) {
            Stall(1)
            And(SM00, 0x02, Local0)}

        // DB00 and DB01 are defined as global buffer fields
        Store(SM05, DB00)           // Store in 1st byte of the buffer
        Store(SM06, DB01)           // Store in 2nd byte of the buffer
        Return(DATW)                // Return word data
} // Method(RWRD)
```

~~1.1.3~~9.3.3 Power Button Support

Physically, the power button is a user push button that switches the system between the working state and the sleeping/soft off state.

- If the system is working when the user presses the power button, this signals the OS to transition to a sleeping/soft off state from the working state.
- If the system is in the sleeping/soft off state when the user presses the power button, the system wakes up and signals the OS to transition to the working state.
- In an emergency situation (for example, the system software is locked up), the user can press the power button for 4 seconds to force the system to go directly to the S5 (fully off) state. The user probably will lose data, but this is used only in an emergency.

9.3.3.1 Power Button Implementation on the Desktop Concept Machine

The power button on the desktop concept machine is implemented as a fixed power button, so no ASL code has to be written to directly support the power button requirement.

The desktop concept machine power button is implemented in a chipset that contains the fixed power button logic shown in Figure 4-8 in section 4.7.2.2.1.1 of the *ACPI Specification, Revision 1*. In particular, the chipset logic

- Generates a PWRBTN event SCI.
- Implements the PWRBTN 4-second override feature.

No additional hardware circuitry needs to be built on the desktop concept machine platform to achieve a power button that complies with the *ACPI Specification, Revision 1*.

~~1.1.1.2~~9.3.3.2 Writing ASL Code that Supports the Desktop Power Button

As stated in the previous section, no ASL code is required to directly support the power button because the desktop concept machine power button is a fixed power button. (If the power button on the concept machine was a control method power button, ASL code would be required to declare a Power Button Device object.)

The block of ASL code that indirectly supports system sleeping and waking states (and the power button) for the Desktop concept machine is shown below

```
Name(\_S0,Package(2){5,5})              // Value to be set in SLP_TYP register (S0)working state
                                                // on this chip set
Name(\_S1,Package(2){4,4})              // Value to be set in SLP_TYP register for S1 state
                                                // on this chip set
Name(\_S2,Package(2){3,3})              // Power on suspend with CPU context lost.
Name(\_S4,Package(2){Zero,Zero})    // Value to be set in SLP_TYP register for Soft Off
                                                // on this chip set
Name(\_S5,Package(2){Zero,Zero})    // Value to be set in SLP_TYP register for Soft Off
                                                // on this chip set
//
//Method(\_PTS) {                           //prepare to sleep(Not used in this implementation)
// }
Method(\WAK,1){
      Notify(THM1,0x80)
}
```

~~1.1.4~~9.3.4 Operation Region and Field Definitions for a Super I/O Chip

The SMC Super I/O chip registers are read and written through a level of indirection. A working register is accessed by designating a particular register in an 8-bit Index register and reading or writing the working register through an 8-bit Data register. The ASL language **OperationRegion**, **Field**, and **IndexField** terms can be declared in combination to define this two-tier register arrangement, assigning field names to the working registers of interest. Once this declaration is made, simple Store terms can be used to read from and write to the field names and the ACPI run-time component built into the OS automatically takes care of all the details of managing the Index and Data registers.

For example, the block of ASL code that declares field names for Super I/O chip working registers is shown below (line numbers are artifacts to make it easier to refer to particular lines of code).

```
1 // Start of Definitions for SMC super I/O device
2     OperationRegion(SMC1,           // name of Operation Region for SuperIO device
3                             SystemIO, // type of address space
4                             0x3F0,          // offset to start of region
5                                             // (default offset for SuperIO device)
6                                             // (Real systems will likely have BIOS relocate this device
7                                             // to avoid conflicts with secondary floppy ID of 0x370)
8                             2)              //size of region in bytes
9     //end of Operation Region
10    Field (     SMC1,                       //fields are in Operation Region named SMC1
11                ByteAcc,
12                NoLock,
13                Preserve)
14                {
15                      INDX,8,               //field named INDX is 8 bits wide
16                      DATA,8                //field named DATA is 8 bits wide
17                }
18    IndexField(INDX,                  //index name
19                      DATA,                 //name of I/O port
20                      ByteAcc,
21                      NoLock,
22                      Preserve)
23                      {
24                            Offset(2),
25                            CFG,8,          //global config control register
26                            Offset(7),
27                            LDN,8,          //Logical Device Number, offset 0x07
28                            Offset(0x30),
29                            ACTR,8,         //activate register, offset 0x30
30                            Offset(0x60),
31                            IOAH,8,         //base I/O addr, offset 0x60
32                            IOAL,8,
33                            Offset(0x70),
34                            INTR,8,         //IRQ, Offset 0x70
35                            Offset(0x72),
36                            INT1,8,         //Second IRQ for some devices, Offset 0x72
37                            Offset(0x74),
38                            DMCH,8,         //DMA channel, offset 0x74
39                            Offset(0xC0),
40                            GP40,8,         //Fast IR control bits, Offset(0xC0)
41                            Offset(0xF0),
42                            OPT1,8,         //Option register 1,  Offset(0xF0)
43                            OPT2,8,         //Option register 2
44                            OPT3,8          //Option register 3
45                      } //end of indexed field
46    Method(ENFG,0){                 // Enter Config Mode for SMC
47          Store(0x55,INDX)
48          Store(0x55,INDX)
49    } // end ENFG method
50    Method(EXFG,0){                 // Exit Config Mode for SMC
51          Store(0xAA,INDX)
52    } // end EXFG method
```

~~1.4~~9.4 Desktop Sample ASL Code

To get a complete listing of the Desktop concept machine sample code, locate the file Dt_smp.asl on the ACPI Website at http://www.teleport.com/~acpi. Load the file Dt_smp.asl into your programming editor and print out the listing.

Notice that the ASL code for the Desktop concept machine has been broken into sections using the ASL compiler **Include** directive. To print out all the Desktop concept machine sample code, you will have to load each of the *.asl files found in the same subdirectory as Dt_smp.asl into your programming editor and print them out.

The ASL code in the file Dt_smp.asl, and in all included files, compiles without errors or warnings using version 1.0.3 of the ASL compiler (asl.exe) distributed by Microsoft.

Note: If there is a difference between the ASL code in the file Dt_smp.asl (and its Include files) and the snippets of sample code shown earlier in this chapter to illustrate concepts, the code in Dt_smp.asl (and its Include files) is more recent than, and overrides, the snippet of code shown in this chapter.

10. ACPI Server Concept Machine

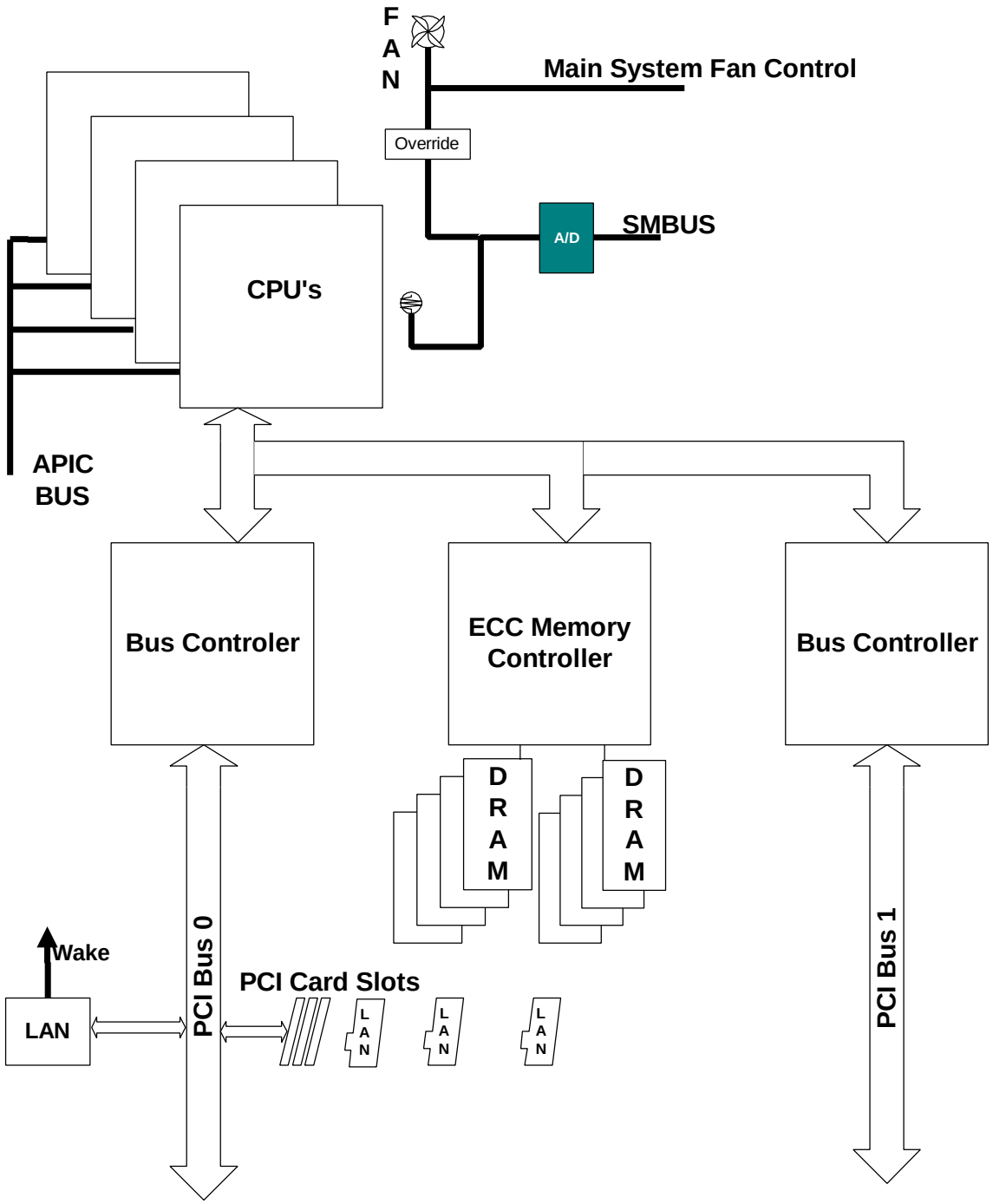
This section presents the ACPI server concept machine. The hardware components of the server concept machine are described by a series of hardware block diagrams. The ACPI name space that models the hardware block diagram is shown, along with sample ASL code that implements the objects in the ACPI name space.

~~1.1~~10.1 Overview of the Server Concept Machine Design

The server concept machine can be characterized as follows:

- Four processors.
- Dual root bridges.
- ACPI hardware support built into the chipset.
- Wake events can come from the LAN controller or the modem.
- Implements the following power saving states:
- System states S1, S4, and S5.
- Processor states C0, C1, and C2.
- Device states D0 and D3. The most prominent device state feature is that SCSI devices are swappable without system power down.

The relationships between the server concept machine components are illustrated in the following simplified block diagram:


F
A
N
Main System Fan Control
Override
A/D
SMBUS
CPU's
APIC
BUS
Bus Controler
ECC Memory
Controller
Bus Controller
D
R
A
M
D
R
A
M
Wake
LAN
PCI Bus 0
PCI Card Slots
L
A
N
L
A
N
L
A
N
PCI Bus 1

The prominent hardware components in the server concept machine design are described in the following table.

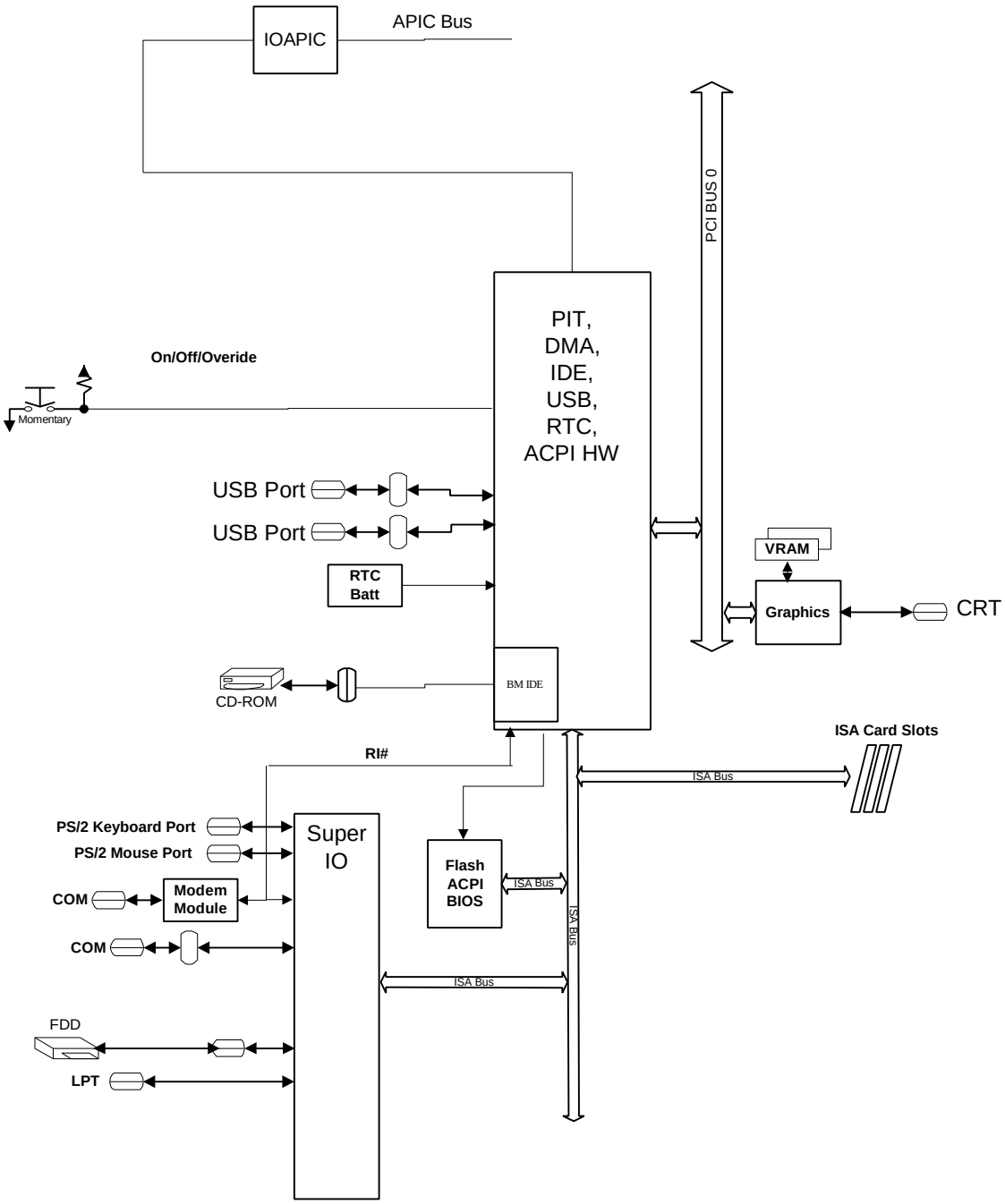
Device	Description
Chipset	• Northbridge - Fictional • Southbridge - Intel
Super I/O	National PC87307VUL
Video	C&T 65548 Flat Panel/CRT GUI Accelerator
LAN	A LAN chipset is included on the motherboard and supports a wake-up function. Additional LAN cards which cannot wake the system are added as plug in cards on the PCI bus.
Hard Drives	Supported via SCSI on the motherboard. Two separate controllers are used. One controller is used to support the internal drives. The second controller supports drives in an external drive cabinet. Drives are hot swappable. Locking mechanisms are used to control the hot swap function.
CD	Connected through the built in IDE controller on the chipset.
Modem	Standard modem chip set interfaced through one of the serial ports. Ring Indicate wired direct to RI# wake up input on the chip set.
A/D	National LM75 Standard device with an I2C output connected through SMB to Embedded controller.
USB	Controller with two ports built into chip set.
Embedded Controller	Used for thermal management, voltage monitoring and tampering alarm.

~~1.2~~10.2 Server Block Diagrams

This section describes the server concept machine with a series of block diagrams. The block diagrams show:

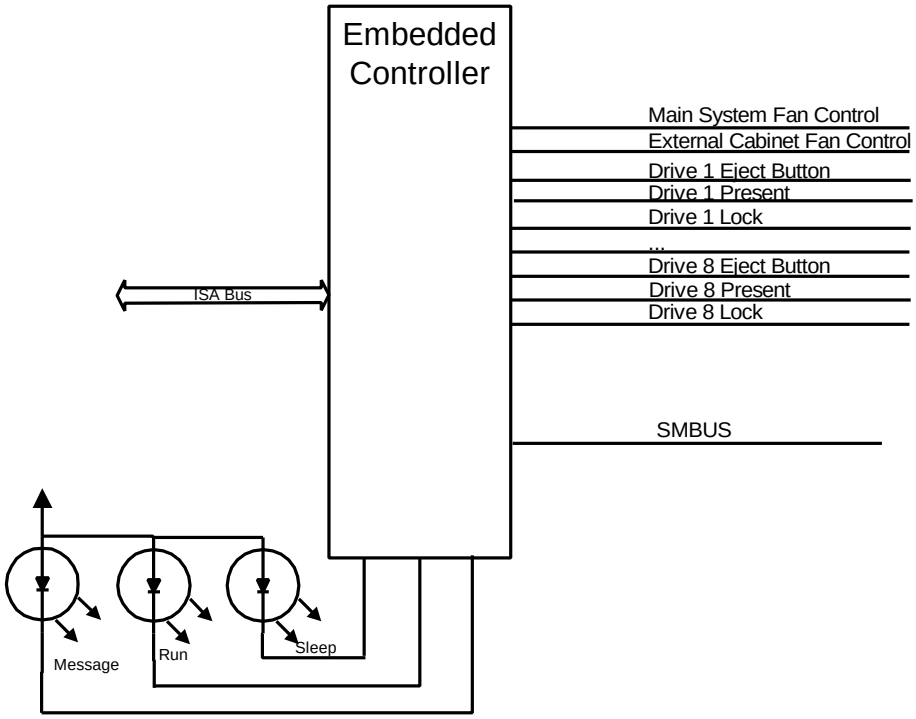
- The relationships between all the components on the server concept machine.
- The relationship of the server hardware components and the embedded controller that is part of the system.
- The components of the server removable drive subsystem in more detail.
- The hardware details of the hot swap drive interface.

The immediately following block diagram is the one that shows the relationships between all the components on the server concept machine.



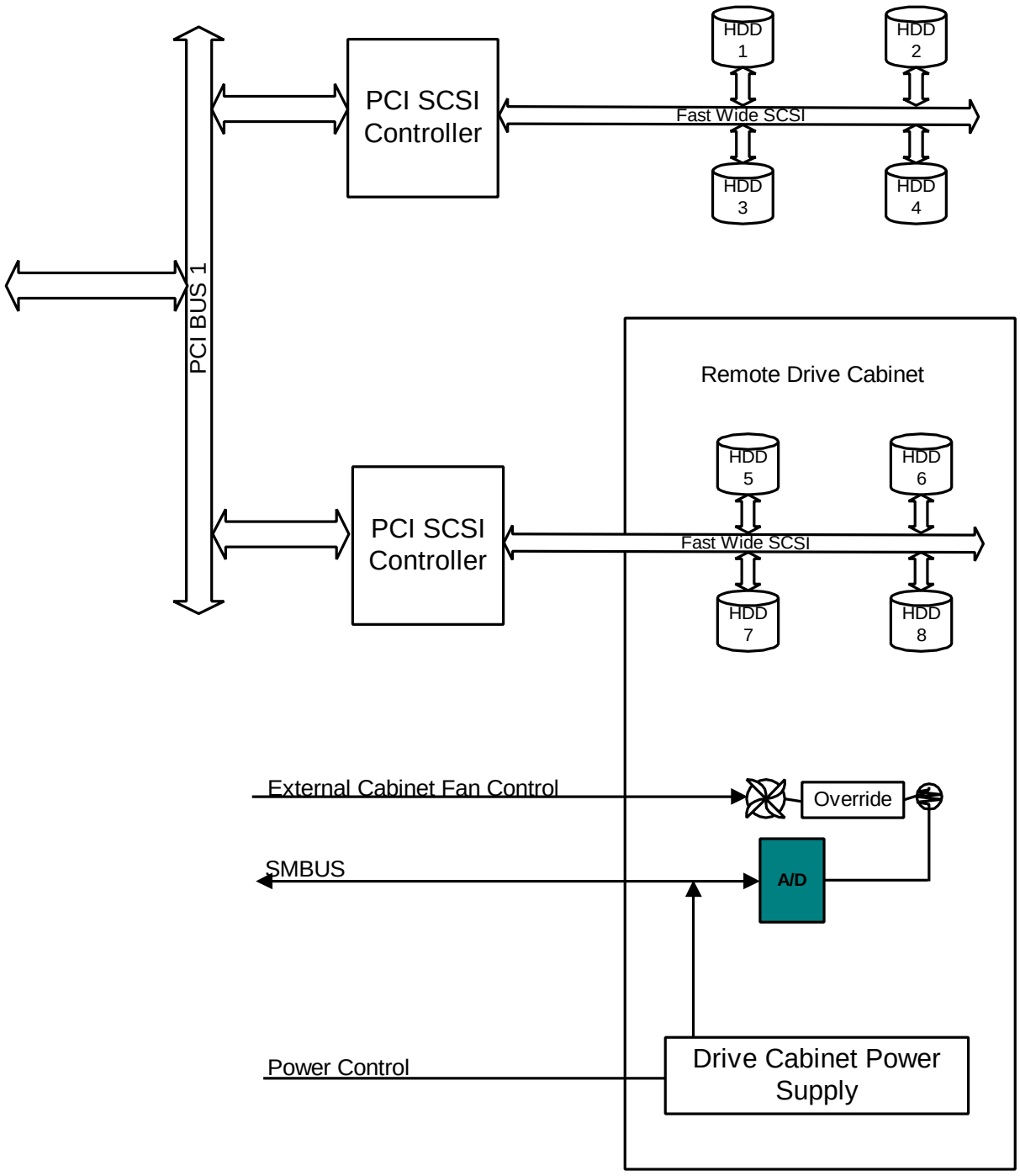
~~1.1.1~~10.2.1 Embedded Controller Details

The following diagram shows the connections between server hardware components and the system embedded controller.



~~1.1.2~~10.2.2 Removable Drive Details

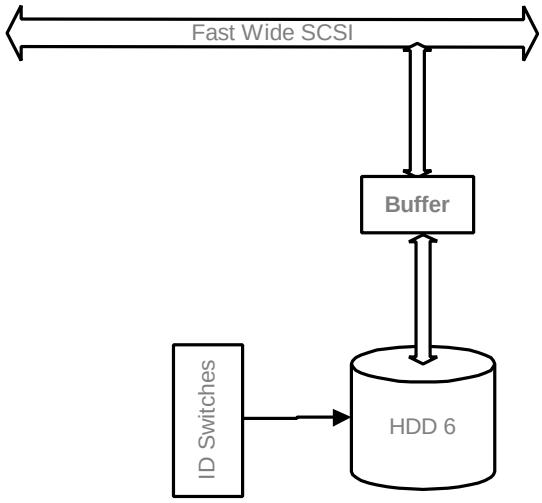
The following figure shows the components that make up the removable drive subsystem on the server concept machine.

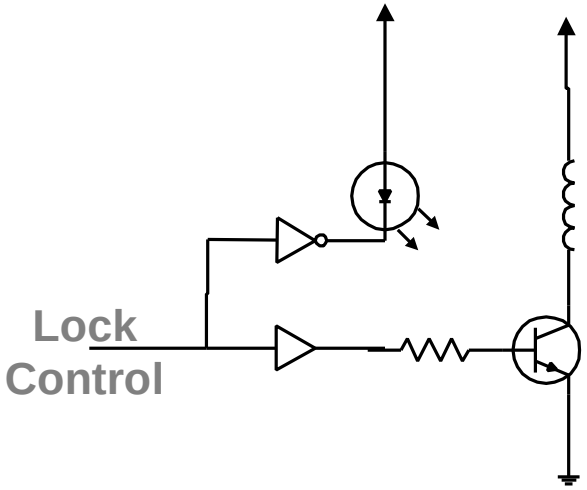


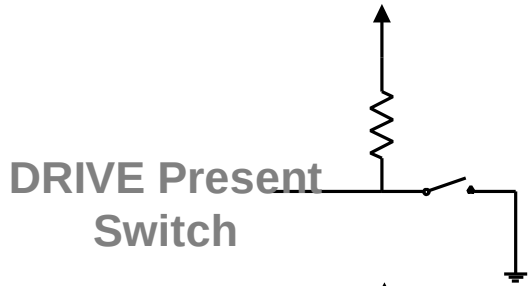
~~1.1.3~~10.2.3 Hot Swap Interface Details

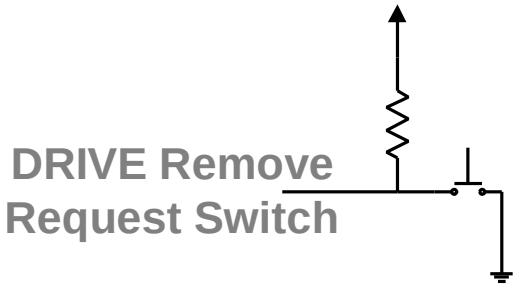
The following figure shows selected details of the hot swap interface. Each drive bay automatically straps the drive ID select bits when the drive is inserted. The ID corresponds to the slot number. Individual drives are isolated from the SCSI bus through a non-glitching buffer. Each drive has a remove request momentary switch. This generates an event through the microcontroller which signals the OS that the user wishes to

remove a drive. A mechanical lock mechanism prevents removal of a drive until the OS acknowledges that it is ready through the microcontroller. An indicator shows when the lock has been released.


Fast Wide SCSI
Buffer
ID Switches
HDD 6


Lock
Control


DRIVE Present
Switch


DRIVE Remove
Request Switch

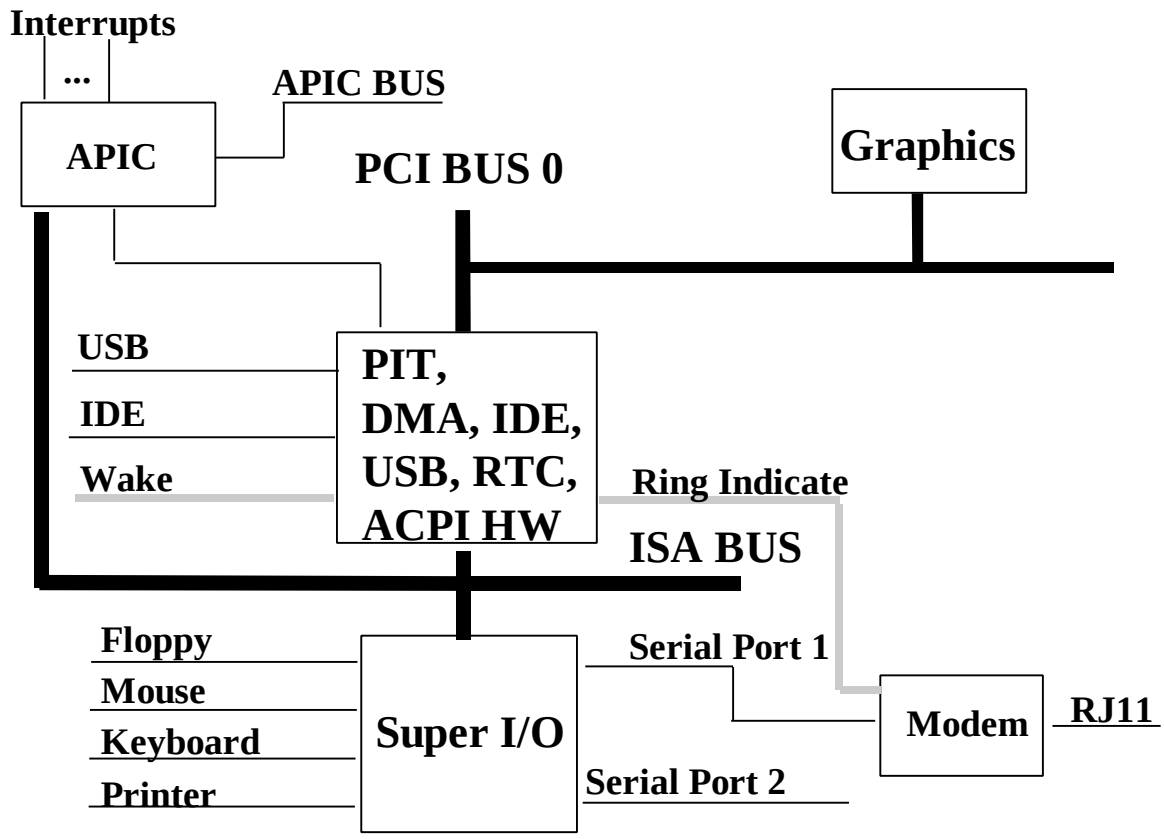
~~1.3~~10.3 Implementation Examples from the Server Concept Machine

This section uses blocks of example ASL code, name space diagrams, and hardware component block diagrams to discuss in some detail the following aspects of the Server concept machine implementation:

- PCI interrupt routing.
- Managing multiple removable hard drives.
- Operation region and field definitions for a Super IO chip.

10.3.1 PCI Interrupt Routing

The following block diagram shows the hardware components involved in PCI interrupt routing.



The following example ASL code sketches an implementation of PCI interrupt routine on the server concept machine. For a more developed block of ASL code that implements PCI interrupt routing, see section 6 of the *Guide*.

```
Scope(\_SB) {                                   // Start of _SB
        .
        .
        .
    Device(LNKA){
        Name(_HID, EISAID("PNP0C0F"))           // PCI interrupt link
        Name(_UID, 1)
        Name(_PRS, ResourceTemplate(){
            Interrupt(ResourceProducer,...) {10,11}     // IRQs 10,11
        })
        Method(_DIS) {...}
        Method(_CRS) {...}
        Method(_SRS, 1) {...}
    }
    Device(LNKB){
        Name(_HID, EISAID("PNP0C0F"))           // PCI interrupt link
        Name(_UID, 2)
        Name(_PRS, ResourceTemplate(){
            Interrupt(ResourceProducer,...) {11,12}     // IRQs 11,12
        })
        Method(_DIS) {...}
        Method(_CRS) {...}
        Method(_SRS, 1) {...}
    }
    Device(LNKC){
        Name(_HID, EISAID("PNP0C0F"))           // PCI interrupt link
        Name(_UID, 3)
        Name(_PRS, ResourceTemplate(){
            Interrupt(ResourceProducer,...) {12,13}     // IRQs 12,13
        })
        Method(_DIS) {...}
        Method(_CRS) {...}
        Method(_SRS, 1) {...}
    }
    Device(LNKD){
        Name(_HID, EISAID("PNP0C0F"))           // PCI interrupt link
        Name(_UID, 4)
        Name(_PRS, ResourceTemplate(){
            Interrupt(ResourceProducer,...) {13,14}     // IRQs 13,14
        })
        Method(_DIS) {...}
        Method(_CRS) {...}
        Method(_SRS, 1) {...}
    }

    Device(PCI0) {                              // Root PCI Bus
        .
        .
        .
        Name(_PRT, Package{
            Package{0x0004ffff, 0, LNKA, 0},  // Slot 1, INTA
            Package{0x0004ffff, 1, LNKB, 0},  // Slot 1, INTB
            Package{0x0004ffff, 2, LNKC, 0},  // Slot 1, INTC
            Package{0x0004ffff, 3, LNKD, 0},  // Slot 1, INTD
            Package{0x0005ffff, 0, LNKB, 0},  // Slot 2, INTA
            Package{0x0005ffff, 1, LNKC, 0},  // Slot 2, INTB
            Package{0x0005ffff, 2, LNKD, 0},  // Slot 2, INTC
            Package{0x0006ffff, 3, LNKA, 0},  // Slot 2, INTD
            Package{0x0006ffff, 0, LNKC, 0},  // Slot 3, INTA
            Package{0x0006ffff, 1, LNKD, 0},  // Slot 3, INTB
            Package{0x0006ffff, 2, LNKA, 0},  // Slot 3, INTC
            Package{0x0006ffff, 3, LNKB, 0},  // Slot 3, INTD
        })
    }
```

~~1.1.2~~10.3.2 Managing Multiple Removable Hard Drives

This section walks through the parts of the server ACPI name space that contain objects used to manage multiple removable hard drives and the ASL code these objects contain. For information about the hardware components involved in managing multiple removable hard drives, see an earlier section.

~~1.1.3~~10.3.3 Operation Region and Field Declarations for a Super I/O Chip

The National Super I/O chip registers are read and written through two levels of indirection. The ASL language **OperationRegion**, **IndexField**, and **BankField** terms can be declared in combination to define this three-tier register arrangement, assigning index field names to the working registers of interest. Once this declaration is made, simple **Store** terms can be used to read from and write to the index field names and the ACPI run-time component built into the OS automatically takes care of all the details of managing the levels of indirection.

For example, the block of ASL code that declares index field names for the National Super I/O chip working registers is shown below.

```
        OperationRegion(N307, SystemIO, 0x2E, 0x2)
        Field(N307, ByteAcc, NoLock, Preserve) {
                NIDX, 8,
                NDAT, 8
        }
        // Define the Index/Data pair for the National 307 chip
        IndexField(NIDX, NDAT, ByteAcc, NoLock, Preserve){
                Offset(0x7),
                LDNM, 8
        }
BankField(N307, LDNM, 0, ByteAcc, NoLock, Preserve) {
                IDX0, 8
        }
        BankField(N307, LDNM, 1, ByteAcc, NoLock, Preserve) {
                IDX1, 8
        }
        BankField(N307, LDNM, 2, ByteAcc, NoLock, Preserve) {
                IDX2, 8
        }
        BankField(N307, LDNM, 3, ByteAcc, NoLock, Preserve) {
                IDX3, 8
        }
        BankField(N307, LDNM, 4, ByteAcc, NoLock, Preserve) {
                IDX4, 8
        }
        BankField(N307, LDNM, 5, ByteAcc, NoLock, Preserve) {
                IDX5, 8
        }
        BankField(N307, LDNM, 6, ByteAcc, NoLock, Preserve) {
                IDX6, 8
        }
//  Encoding of field names for National 307 configuration Space
//  AxCC
//      Where, A is name of device in super I/O
//              K - Keyboard
//              L - Parallel Port
//              U - UART
//              F - Floppy Disk
//              S - System Logic
//  Where x is the number of the device (1-2)
//  And CC is a description of the field for that device
//              AD - Address
//              Ix - Interrupt configuration
//              EN - Enable
//              DM - DMA
//              PD - Power Down
//              Fx - Float
//              Cx - Configuration register of some sort

        IndexBankField(IDX0N307, NDATKBC,0, ByteAcc, NoLock, Preserve) {
        //  The N307 supports configuration of the keyboard controller
        //  However most systems only support port 0x62-0x66

                Offset(0x30),
                KEN,    8,              // Enable/Disable the keyboard
                Offset(0x60),
                KAD0,   8,              // KBC data port (15-8)
                KAD1,   8,              // KBC data port (7-0)
                KAD2,   8,              // KBC command port (15-8)
                KAD3,   8,              // KBC command port (7-0)
                Offset(0x70),
                KIR0,   8,              // IRQ Pin
                KIR1,   8,              // IRQ type
                }

        IndexBankField(IDX1N307, NDATMSE,1, ByteAcc, NoLock, Preserve) {
        // Allows configuration of the PS/2 mouse interrupt, normally IRQ12
                Offset(0x30),
                MEN,    8,              // Mouse Enable
                Offset(0x70),
                MIR0,   8,              // IRQ Pin
```

```
        MIR1,   8               // IRQ type
        }

    IndexBankField(IDX2N307, NDATRTC, 2, ByteAcc, NoLock, Preserve) {
    // Allows configuration of the RTC
        Offset(0x30),
        REN,    1,              // RTC Enable
        Offset(0x60),
        RAD0,   8,              // RTC Base (15-8)
        RAD1,   8,              // RTC Base (7-0)
        Offset(0x70),
        RIR0,   8,              // RTC Interrupt Pin
        RIR1,   8,              // RTC Interrupt Type
        }

    IndexBankField(IDX3N307, NDATFDC, 3, ByteAcc, NoLock, Preserve) {
    // Allows configuration of I/O, IRQ and DMA.  Additionally has a float control
    // For FDC pins to allow powering off drive.  Other config registers are statically
    // programmed by BIOS prior to handoff to OS

        Offset(0x30),
        FEN, 8,             // FDC Enable
        Offset(0x60),

        FAD0,   8,              // FDC Base (15-8)
        FAD1,   8,              // FDC Base (7-0)
        Offset(0x70),

        FIR0,   8,              // FDC Interrupt Pin
        FIR1,   8,              // FDC Interrupt type
        Offset(0x74),

        FDM,    8,              // DMA Select
        Offset(0xF0),
        FFL, 1              // FDC Pin Float enable
        }

    IndexBankField(IDX4N307, NDATPRT, 4, ByteAcc, NoLock, Preserve) {
        Offset(0x30),
        PEN, 8,             // Parallel Port Enable
        Offset(0x60),
        PAD0,   8,              // PRT Base (15-8)
        PAD1,   8,              // PRT Base (7-0)
        Offset(0x70),
        PIR0,   8,              // PRT Interrupt Pin
        PIR1,   8,              // PRT Interrupt Type
        Offset(0x74),
        PDM,    8,              // PRT DMA Select
        Offset(0xF0),
        // Bits for power management control
        PFL, 1,             // PRT Float
        PCEN,   1,              // PRT Clock Enable
        NULL,   3,
        PMOD,   2               // Selects mode of parallel port
        }

    IndexBankField(IDX5N307, NDATUARA, 5, ByteAcc, NoLock, Preserve) {
        Offset(0x30),
        UAEN,   8,              // UART A/SIR Enable
        Offset(0x60),
        UAA0,   8,              // UART A Base (15-8)
        UAA1,   8,              // UART A Base (7-0)
        Offset(0x70),
        UAI0,   8,              // UART A Interrupt Select
        UAI1,   8,              // UART A Interrupt Type
        Offset(0x74),
        UAD0,   8,              // UART A Primary DMA Select
        UAD1,   8,              // UART A Secondary DMA Select
        Offset(0xF0),
        UAFL,   1,              // UART A Float Control
        UAPM,   1,              // 0-RI wake-up IRQ/clocks enabled,
```

```
                        // 1-Clocks on
    NULL,     5,
    UAEX,     1         // Enables the extended banks to be accessed
    }

IndexBankField(IDX6N307, NDATUARB, 6, ByteAcc, NoLock, Preserve) {
    Offset(0x30),
    UBEN,     8,        // UART B Enable
    Offset(0x60),
    UBA0,     8,        // UART B Base (15-8)
    UBA1,     8,        // UART B Base (7-0)
    Offset(0x70),
    UBI0,     8,        // UART B Interrupt Select
    UBI1,     8,        // UART B Interrupt Type
    Offset(0xF0),
    UBFL,     1,        // UART B Float Control
    UBPM,     1,        // 0-RI wake-up IRQ/clocks enabled,
                        // 1-Clocks on
    NULL,     5,
    UBEX,     1         // Enables the extended banks to be accessed
    }
```

1.410.4 Server Concept Machine Sample ASL Code

The following ASL code implements the ACPI name space for the server concept machine. Notice that the ASL code has been broken into sections using the ASL compiler **Include** directive. The following sections of the server concept machine sample code is organized in the same way.

1.510.5 Server Concept Machine ACPI Name Space

This section shows the ACPI name space and ASL code that creates that name space, based on the platform design illustrated in the previous section.

```
\_PR
      CPU0
      CPU1
      CPU2
      CPU3
\_PTS
\_S0
\_S1
\_S4
\_S5
\_WAK
\_SI                    //System Indicator
      _MSG
      _SST

\_TZ                    //Thermal Zone
      PFN0                    //Power Resource PFN0
            _STA              //Status
            _ON               //Fan On
            _OFF              //Fan Off
      PFN1                    //Power Resource PFN1
            _STA              //Status
            _ON               //Fan On
            _OFF              //Fan Off
      FAN0
            _HID              //Hardware Device ID
            _PR0              // power resource
      FAN1
            _HID              //Hardware Device ID
            _PR0              // power resource
      THM0                    //Thermal Zone for main system
            _TMP              //Get Current temp
            _AC0              //Active cooling trip point x
            _AL0              //Active cooling device list  (e.g. FAN0)
            _CRT              //Critical Temp.
      THM1                    //Thermal Zone for remote drive cabinet
            _TMP              //Get Current temp
            _AC0              //Active cooling trip point x
            _AL0              //Active cooling device list  (e.g. FAN1)
            _CRT              //Critical Temp.
\_GPE
      _L00              //GP event from embedded controller
\_SB
      LNKA
      LNKB
      LNKC
      LNKD
      PCI0
            _HID              //Hardware ID
            _ADR              //Device address on the PCI bus
            _CRS              //Current Resource (Bus number 0)
            _PRT
      PX40
            _ADR
      USB0
            _ADR
            _STA
            _PRW
      PX43
            _ADR
      ISA
            _HID
            EC0               //Embedded controller for system management functions
                  _HID        //Hardware device ID for embedded controller
                  _CRS
                  _Q7         //Thermal Event Notification main cabinet
                  _Q8         //Thermal Event Notification remote cabinet
                  _Q11        //Drive 1 remove request
                  _Q12        //Drive 2 remove request
                  _Q13        //Drive 3 remove request
                  _Q14        //Drive 4 remove request
```

```
        _Q15            //Drive 5 remove request
        _Q16            //Drive 6 remove request
        _Q17            //Drive 7 remove request
        _Q18            //Drive 8 remove request
        _Q21            //Drive 1 insertion
        _Q22            //Drive 2 insertion
        _Q23            //Drive 3 insertion
        _Q24            //Drive 4 insertion
        _Q25            //Drive 5 insertion
        _Q26            //Drive 6 insertion
        _Q27            //Drive 7 insertion
        _Q28            //Drive 8 insertion
    RTC
        _HID
        _CRS
        _PRW
    COPR                //Coprocessor device
        _HID            //Hardware Device ID
        _CRS            //Current Resource
    DMA                     //DMA Device
        _HID            //Hardware Device ID
        _CRS            //Current Resource
    PIC                 //Interrupt controller device
        _HID            //Hardware Device ID
        _CRS            //Current Resource
    MEM                     //Memory Device
        _HID            //Hardware Device ID
        _CRS            //Current Resource
    SPKR                //Speaker Device
        _HID            //Hardware Device ID
        _CRS            //Current Resource
    TMR                 //Timer Device
        _HID            //Hardware Device ID
        _CRS            //Current Resource
    EIO
    PS2M                //PS2 Mouse Device
        _HID            //Hardware Device ID
        _STA            //Status of the PS2 Mouse device
        _CRS            //Current Resource
    PS2K                //PS2 Keyboard Device
        _HID            //Hardware Device ID
        _CRS            //Current Resource
        _STA            //Status of PS2 Keyboard device
    FDC0                //Floppy Disk controller
        _HID            //Hardware Device ID
        _STA            //Status of the Floppy disk controller
        _DIS        //Disable
        _CRS            //Current Resource
        _PRS            //Possible Resource
        _SRS            //Set Resource
    LPT                 //Standard Printer
        _HID            //Hardware Device ID
        _STA            //Status of the printer device
        _DIS        //Disable
        _CRS            //Current Resource
        _PRS            //Possible Resource
        _SRS            //Set Resource
    ECP                 //Extended capabilities Port
        _HID            //Hardware Device ID
        _STA            //Status of the port device
        _DIS        //Disable
        _CRS            //Current Resource
        _PRS            //Possible Resource
        _SRS            //Set Resource
    UARA                //Communication Device (Modem Port)
        _HID            //Hardware Device ID
        _STA            //Status of the Communication Device
        _DIS        //Disable
        _CRS            //Current Resource
        _PRS            //Possible Resource
        _SRS            //Set Resource
```

```
                _PRW            //Wake-up control method
            COMB
                _HID            //Hardware Device ID
                _STA            //Status of the COM device
                _DIS        //Disable
                _CRS            //Current Resource
                _PRS            //Possible Resource
                _SRS            //Set Resource

//  IDE and Video don't appear as there is no value added hardware and
//  system uses the standard PCI PnP and standard driver support power
//  management

        LAN0                    //LAN Device on motherboard
            _ADR                //Hardware Device ID
            _PRW                //Wake-up

// Note: The following name space objects do not have matching ASL code
//       in this revision of the "ACPI Implementers Guide"

\PCI1
            _HID                //Hardware Device ID
            _ADR                //Device address
            _CRS                //Current Resource (Bus number 1)
        SCS0                    //SCSI controller
            _ADR                //Device address
            HDD1                //SCSI Hard drive device
                _STA            //Status of device
                _IRC        //Inrush Current
                _LCK            //Lock control
            HDD2                //SCSI Hard drive device
                _STA            //Status of device
                _IRC        //Inrush Current
                _LCK            //Lock control
            HDD3                //SCSI Hard drive device
                _STA            //Status of device
                _IRC        //Inrush Current
                _LCK            //Lock control
            HDD4                //SCSI Hard drive device
                _STA            //Status of device
                _IRC        //Inrush Current
                _LCK            //Lock control
        SCS1                    //SCSI controller
            _ADR                //Device address

            HDD5                //SCSI Hard drive device
                _STA            //Status of device
                _IRC        //Inrush Current
                _LCK            //Lock control

            HDD6                //SCSI Hard drive device
                _STA            //Status of device
                _IRC        //Inrush Current
                _LCK            //Lock control
            HDD7                //SCSI Hard drive device
                _STA            //Status of device
                _IRC        //Inrush Current
                _LCK            //Lock control
            HDD8                //SCSI Hard drive device
                _STA            //Status of device
                _IRC        //Inrush Current
                _LCK            //Lock control
```

~~1.6~~10.6 Server Sample ASL Code

To get a complete listing of the Server concept machine sample code, locate the Srv_exa1.asl on the ACPI Website at http://www.teleport.com/~acpi. Load the file Srv_smp.asl into your programming editor and print out the listing.

Notice that the ASL code for the Desktop concept machine has been broken into sections using the ASL compiler **Include** directive. To print out all the Server concept machine sample code, you will have to load each of the *.asl files found in the same subdirectory as Srv_smp.asl into your programming editor and print them out.

The ASL code in the file Srv_smp.asl, and in all included files, compiles without errors or warnings using version 1.0.3 of the ASL compiler (asl.exe) distributed by Microsoft.

Note: If there is a difference between the ASL code in the file Srv_smp.asl (and its Include files) and the snippets of sample code shown earlier in this chapter to illustrate concepts, the code in Srv_smp.asl (and its Include files) is more recent than, and overrides, the snippet of code shown in this chapter.

.

11. Using the ACPI Embedded Controller and SMBus Interfaces

Section 13 of the *ACPI Specification, Revision 1.0*, specifies and embedded controller interface and an SMBus interface that is emulated through a block of registers in the embedded controller interface. This section contains several examples that illustrate the use of these interfaces.

An embedded controller is not required in an ACPI-compatible system, but embedded controllers are motherboard components on many systems being built today, especially mobile systems. If an ACPI-compatible system does use an embedded controller, the benefit of using the embedded controller interface specification in the *ACPI Specification, Revision 1.0*, is that the interface between an ACPI-compatible OS and the embedded controller is standardized across operating systems from different vendors and across different versions of operating systems from the same vendor. Using the specified interface also enables an ACPI-compatible OS to interact directly with the embedded controller as the OS carries out its power, thermal, and other event management policies.

Use of the specified SMBus controller interface that is emulated by the embedded controller is also optional. Benefits of using this interface are:

- The interface between the OS and the SMBus controller on an ACPI-compatible platform is standardized (in contrast to the many different interfaces that are offered to by the SMBus controllers that are currently on the market). Using a standardized SMBus controller interface makes it easier for the OS to interact directly with devices on the SMBus as the OS carries out its power, thermal, and other event management policies.
- Since the SMBus controller interface specified in the *ACPI Specification, Revision 1.0*, is part of the embedded controller, the embedded controller can filter commands sent over the SMBus in order to increase platform security.

11.1 Embedded Controller Example #1

In this example, the OS is attempting to set one bit in an eight-bit register in embedded controller space. This is accomplished through a read, modify, write sequence. During this sequence, the embedded controller detects an event that requires a notification. This example shows how the interrupt arbitration is handled.

The following are the specifics of this particular implementation::

- EC_SC port address is 68h in system I/O space.
- EC_DATA port address is 69h in system I/O space.

The OS is attempting to set E32.3 (address 32h, embedded controller space, bit three). The embedded controller notices that the system has detected a critical thermal event during the read/write process which initiates an OS thermal notification process.

Host processor sequence	Embedded controller sequence
OS checks for IBF to be cleared I/O RD 0x68→0x00	
Write BE_EC command byte to controller: I/O WR 0x68→0x82 (command byte for burst enable).	Embedded controller receives the input buffer full interrupt and reads input buffer: RD IBR→0x82

Host processor sequence	Embedded controller sequence
OS schedules other tasks and waits for an SCI event.EC_SCI to be deasserted.	Embedded controller receives burst enable command and sets the BURST bit in status register when ready. It also puts the Burst Acknowledge byte (0x90) into the SCI output buffer, sets the OBF bit, and generates an SCI to signal the OS that it is in BURST mode.
SCI received.	Embedded controller waits for input buffer full.
OS checks for output buffer full: I/O RD 0x68→0x11 (bit 4 is set meaning OBF=1).	Embedded controller waits for input buffer full.
OS reads output buffer: I/O RD 0x69→0x90 (Burst Acknowledge byte)	Embedded controller waits for input buffer full.
OS writes RD_EC command byte to controller: I/O WR 0x68→0x80 (command byte for read).	Embedded controller waits for input buffer full.
OS waits 1 microsecond.	IBF is set.
OS waits for SCI to signal IBF=0	Embedded controller reads input buffer to get command: RD IBR→0x80
OS waits for SCI to signal IBF=0	Embedded controller dispatches command 80h handler.
OS waits for SCI to signal IBF=0	Embedded controller drives SCI to signal that IBF=0 (input buffer is empty).
OS checks status register to continue processing: I/O RD 0x68→0x10 (burst enabled, input buffer empty)	Embedded controller waits for input buffer full.
OS sends next byte of command sequence (address): I/O WR 0x69→0x32 (address byte to read)	Embedded controller waits for input buffer full.
OS waits 1 microsecond	Embedded controller reads next byte: RD IBR→0x32
OS waits for SCI to signal OBF=1	Embedded controller reads at location 0x32: RD 0x32→0xA6
OS waits for SCI to signal OBF=1	Embedded controller writes data to output buffer: WR OBR→0xA6 and drives SCI to signal that OBF=1 (output buffer full).
OS checks for output buffer full: I/O RD 0x68→0x11 (bit 0 is set meaning OBF)	Embedded controller waits for input buffer full.

Host processor sequence	Embedded controller sequence
OS reads returned data: I/O RD 0x69→0xA6	Embedded controller waits for input buffer full.
OS sets bit 3 (0xA6→0xAE) and prepares to generate write command.	Embedded controller detects critical thermal event (high priority) and sets SCI_EVT which generates an SCI.
OS checks status register: I/O RD 0x68→0x30 (burst enabled, event detected, input buffer empty).	Embedded controller waits for input buffer full.
OS writes QR_EC command byte to controller. I/O WR 0x68→0x84 (command byte for query).	Embedded controller waits for input buffer full.
OS waits for 1 microsecond.	Embedded controller waits for input buffer full.
OS waits for SCI to signal OBF=1.	Embedded controller receives query command and returns the notification header for critical thermal event (in this case, 0x1A). Embedded controller resets event flag. WR OBR→0x1A (generates SCI to signal OBF=1) WR STR→0x11
OS checks status register for returned query data: I/O RD 0x68→0x11 (burst enabled, output buffer full)	Embedded controller waits for input buffer full.
OS reads returned data from output buffer: I/O RD 0x69→0x1A (OS now has source of query and can process the event)	Embedded controller waits for input buffer full.
OS checks status register to continue processing: I/O RD 0x68→0x10 (burst enabled, input buffer empty).	Embedded controller waits for input buffer full.
OS writes WR_EC command byte to controller: I/O WR 0x68→0x80 (command byte for write)	Embedded controller receives write command. RD IBR→0x80
OS waits for SCI to signal IBF=0.	Embedded controller drives SCI to signal IBF=0.
OS checks status register to continue processing: I/O RD 0x68→10h (burst enabled, input buffer empty)	Embedded controller waits for IBF to receive the remaining portion of the write command.
OS sends next byte of command sequence (address): I/O WR 0x69→0x32 (address byte to write)	Embedded controller receives address byte: RD IBR→0x32
OS waits 1 microsecond.	Embedded controller drives SCI to signal IBF=0.

Host processor sequence	Embedded controller sequence
OS waits for SCI to signal IBF=0.	Embedded controller waits for input buffer full.
OS checks status register to continue processing: I/O RD 0x68→0x10 (burst enabled, input buffer empty)	Embedded controller waits for input buffer full.
OS writes last byte of command: I/O WR 0x69→0xAE (data byte to write)	Embedded controller reads next byte: RD IBR→0xAE
OS waits 1 microsecond.	Embedded controller drives SCI to signal IBF=0.
OS waits for SCI to signal IBF=0.	Embedded controller writes data: WR 0x32→0xAE
OS checks status register to continue processing: I/O RD 0x68→0x10 (burst enabled, input buffer empty)	Embedded controller waits for input buffer full.
OS writes BD_EC command byte to controller: I/O WR 0x68→0x83 (command byte for burst disable)	Embedded controller waits for input buffer full.
OS schedules other tasks.	Embedded controller receives burst disable command and resumes normal processing: WR STR→00h (no longer bursting)

~~1.2~~11.2 Embedded Controller Example #2

This is an example of reading status from a smart charger by using the SMBus host interface. In this example, the host sends the SMB address, command, and the command protocol to an SMBus controller implemented inside of an embedded controller. The controller returns two bytes of data (charger status).

The following are the specifics of this particular implementation:

- EC_SC port address is 66h in system I/O space.
- EC_DATA port address is 62h in system I/O space.
- SMBus address base is 40h.
- SMBus Command Complete Notification Header is 21h.

The host sends a read smart charger status command, which needs to generate the following SMBus transaction:

- SMB_ADDR (BASE+2=0x42)=Charger Device=0x12
- SMB_CMD (BASE+3=0x43)=ChargerStatus=0x13
- SMB_PRTCL (BASE+0=0x40)=Read Word=0x07

which then returns the following information

- SMB_DATA[0] (BASE+4=0x44)=Low data byte of received charger status=0x10 (for example).
- SMB_DATA[1] (BASE+5=0x45)=High data byte of received charger status=0xC0 (for example).

Host processor sequence	Embedded controller sequence
OS checks for IBF to be cleared: I/O RD 0x66→0x00	
Write BE_EC command byte to controller: I/O WR 0x66→0x82 (command byte for burst enable)	IBF is set.
OS schedules other tasks and waits for an SCI event.EC_SCI to be deasserted.	EC processes the burst enable command, sets the BURST flag, and sets WR STR→0x10 (burst enabled) which generates an SCI when OBF=1.
OS issues write byte command to SMB_ADDR register.	Embedded controller waits for input buffer full.
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty)	Embedded controller waits for input buffer full.
OS writes WR_EC command byte to controller: I/O WR 0x66→0x81 (command byte for write)	Embedded controller receives write command. RD IBR→0x81
OS waits for 1 microsecond.	Embedded controller drives SCI to signal input buffer empty.
OS waits for SCI to signal input buffer empty.	Embedded controller waits for input buffer full.
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty).	Embedded controller waits for input buffer full.
OS writes next byte of command sequence (address of location to write=base+2=0x40+2=0x42): I/O WR 0x62→0x42 (address byte to write)	Embedded controller receives address byte: RD IBR→0x42
OS waits for 1 microsecond.	Embedded controller drives SCI to signal input buffer empty.
OS waits for SCI to signal input buffer empty.	Embedded controller waits for input buffer full.
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty)	Embedded controller waits for input buffer full.
OS writes last byte of command sequence (address of device on SMBus=charger device=0x12). I/O WR 0x62→0x12 (data byte to write).	Embedded controller reads next byte. RD IBR→0x12
OS waits for 1 microsecond and then waits for an SCI event.	Embedded controller drives SCI to signal input buffer empty.

Host processor sequence	Embedded controller sequence
OS issues write byte command to SMB_CMD register.	Embedded controller writes data to SMBus buffer: WR 0x42→0x12 (SMB_ADDR=Charger Device).
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty)	Embedded Controller waits for input buffer full.
OS writes WR_EC command byte to controller: I/O WR 0x66→0x81 (command byte for write)	Embedded controller receives write command. RD IBR→0x81
OS waits 1 microsecond.	Embedded controller drives SCI to signal input buffer empty.
OS waits for SCI to signal input buffer empty.	Embedded controller waits for input buffer full.
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty).	Embedded controller waits for input buffer full to receive the remaining portion of the write command.
OS writes next byte of command sequence (address of location to write=base+3=0x40+3=0x43): I/O WR 0x62→0x43 (address byte to write)	Embedded controller receives address byte. RD IBR→0x43
OS waits for 1 microsecond and then waits for an SCI event.	Embedded controller drives SCI to signal input buffer empty.
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty)	Embedded controller waits for input buffer full.
OS writes last byte of command sequence (command byte for SMBus transaction=ChargerStatus=0x13). I/O WR 0x62→0x13 (data byte to write).	Embedded controller reads next byte: RD IBR→0x13
OS waits for 1 microsecond and then waits for an SCI event.	Embedded controller drives SCI to signal input buffer empty.
OS issues write byte command to SMB_PRTCL register to initiate SMBus transaction.	Embedded controller writes data to SMBus buffer: WR 0x43→0x13 (SMB_CMD=ChargerStatus).
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty)	Embedded controller waits for input buffer full.
OS writes WR_EC command byte to controller: I/O WR 0x66→0x81 (command byte for write)	Embedded controller receives write command: RD IBR→0x81
OS waits 1 microsecond and then waits for an SCI event.	Embedded controller drives SCI to signal input buffer empty.

Host processor sequence	Embedded controller sequence
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty)	Embedded controller waits for input buffer full to receive the remaining portion of the write command.
OS sends next byte of command sequence (address of location to write=base+0=0x40+0=0x40): I/O WR 0x62→0x40 (address byte to write)	Embedded controller receives address byte: RD IBR→0x40
OS waits for SCI to signal input buffer empty.	Embedded controller drives SCI to signal input buffer empty.
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty)	Embedded controller waits for input buffer full.
OS writes last byte of command sequence (protocol byte for SMBus transaction=Read Word=0x07): I/O WR 0x62→0x07 (data byte to write)	Embedded controller reads next byte: RD IBR→0x07
OS waits for SCI to signal input buffer empty.	Embedded controller writes data to SMBus buffer: WR 0x40→0x07 (SMB_PRTCL=Read Word)
OS waits for SCI to signal input buffer empty.	Embedded controller drives SCI to signal input buffer empty.
OS issues burst disable command to embedded controller to allow SMBus transaction to start.	Embedded controller waits for input buffer full.
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty)	Embedded controller waits for input buffer full.
OS writes BD_EC command byte to controller: I/O WR 0x66→0x83 (command byte for burst disable)	Embedded controller burst disable command: RD IBR→0x83
OS clears and re-enables SCI for embedded controller interrupt.	Embedded controller clears burst state in status register: WR STR→0x00 (burst disabled).
OS waits for SCI.	Embedded controller sees SMB_PRTCL register is non-zero and initiates SMBus transaction.
OS waits for SCI.	Embedded controller completes SMBus transaction and sets event flag in status register and generates SCI. WR STR→0x20 (SCI event detected).
OS receives embedded controller SCI.	Embedded controller waits for input buffer full.

Host processor sequence	Embedded controller sequence
OS checks status register to continue processing: I/O RD 0x66→0x20 (event detected, burst disabled, input buffer empty, output buffer empty)	Embedded controller waits for input buffer full.
OS issues query command to embedded controller.	Embedded controller waits for input buffer full.
OS writes QR_EC command byte to controller: I/O WR 0x66→0x84 (command byte for query)	Embedded controller receives input buffer full interrupt and reads command: RD IBR→0x84
OS clears and re-enables SCI for embedded controller interrupt and waits for SCI to signal output buffer full.	Embedded controller writes query notification to output buffer (notification for SMBus complete): WR OBR→0x21
OS waits for SCI to signal output buffer full.	Embedded controller clears event status bit (no more events pending): WR STR→0x01 (no event, output buffer full)
OS waits for SCI to signal output buffer full.	Embedded controller generates SCI.
OS receives SCI and checks status register: I/O RD 0x66→0x01 (output buffer full)	Embedded controller waits for input buffer full.
OS reads returned data from data register: I/O RD 0x62→0x21	Embedded controller waits for input buffer full.
OS executes handler for returned notification (SMBus command complete).	Embedded controller waits for input buffer full.
OS issues burst enable command to prepare for reading of multiple bytes of returned data.	Embedded controller waits for input buffer full.
OS checks for busy bit and for IBF to be cleared I/O RD 0x66→0x00	Embedded controller waits for input buffer full.
Write BE_EC command byte to controller: I/O WR 0x66→0x82 (command byte for burst enable).	Embedded controller receives input buffer full interrupt and reads command: RD IBR→0x82 (burst enable command received).
OS schedules other tasks and waits for an SCI event (EC_SCI to be deasserted).	Embedded controller processes burst enable command, sets burst bit in status register, and generates SCI when ready: WR STR→0x10 (burst enabled)
OS receives embedded controller SCI.	Embedded controller waits for input buffer full.
OS issues read byte command for SMB_DATA[0] register.	Embedded controller waits for input buffer full.

Host processor sequence	Embedded controller sequence
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty).	Embedded controller waits for input buffer full.
OS writes RD_EC command byte to controller: I/O WR 0x66→0x80 (command byte for read).	Embedded controller receives RD_EC command. RD IBR→0x80
OS waits for 1 microsecond and then waits for SCI to signal output buffer full.	Embedded controller drives SCI to signal input buffer empty.
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty).	Embedded controller waits for input buffer full to receive the remaining portion of the RD_EC command.
OS writes next byte of command sequence (address of SMB_DATA[0]=base+4=0x40+4=0x44): I/O WR 0x62→0x44 (address byte to read).	Embedded controller receives address byte. RD IBR→0x44
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty)	Embedded controller reads requested data from buffer: RD 0x44→0x10 (value at SMB_DATA[0])
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty).	Embedded controller writes data to output buffer: WR OBR→0x10 (sets OBF flag)
OS checks status register to continue processing: I/O RD 0x66→0x11 (burst enabled, input buffer full).	Embedded controller waits for input buffer full.
OS reads returned data from data register: I/O RD 0x62→0x10	Embedded controller waits for input buffer full.
OS issues read byte command for SMB_DATA[1] register.	Embedded controller waits for input buffer full.
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty)	Embedded controller waits for input buffer full.
OS writes RD_EC command byte to controller: I/O WR 0x66→0x80 (command byte for read).	Embedded controller receives RD_EC command. RD IBR→0x80
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty)	Embedded controller waits for IBF to receive the remaining portion of the write command.

Host processor sequence	Embedded controller sequence
OS sends next byte of command sequence (address of SMB_DATA[1]=base+5=0x40+5=0x45): I/O WR 0x62→0x45 (address byte to read).	Embedded controller receives address byte: RD IBR→0x45
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty)	Embedded controller reads requested data from buffer: RD 0x45→0xC0 (value at SMB_DATA[1])
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty)	Embedded controller writes data to output buffer: WR OBR→0xC0 (sets OBF flag).
OS checks status register to continue processing: I/O RD 0x66→0x11 (burst enabled, input buffer full)	Embedded controller waits for input buffer full.
OS reads returned data from data register: I/O RD 0x62→0xC0	Embedded controller waits for input buffer full.
OS issues burst disable command to allow embedded controller to resume normal processing.	Embedded controller waits for input buffer full.
OS checks status register to continue processing: I/O RD 0x66→0x10 (burst enabled, input buffer empty).	Embedded controller waits for input buffer full.
OS writes BD_EC command byte to controller: I/O WR 0x66→0x83 (command byte for burst disable)	Embedded controller receives burst disable command: RD IBR→0x83
OS clears and re-enables SCI for embedded controller interrupt.	Embedded controller clears burst flag in status register: WR STR→0x00 (burst disabled)
	Embedded controller resumes normal processing.

12. Definition of Terms

This section contains term definitions from the *ACPI Specification, Revision 1.0*, that are particularly relevant to this *Guide*.

ACPI Name Space:

The ACPI Name Space is a hierarchical tree structure in OS-controlled memory that contains named objects. These objects may be data objects, control method objects, bus/device package objects, etc. The OS dynamically changes the contents of the Name Space at run time by loading and/or unloading definition blocks from the ACPI Tables that reside in the ACPI BIOS. All the information in the ACPI Name Space comes from the Differentiated System Description Table, which contains the Differentiated Definition Block, and one or more other definition blocks.

AML:

ACPI control method ***M***achine ***L***anguage. Pseudocode for a virtual machine supported by an ACPI-compatible operating system and in which ACPI control methods are written.

ASL:

ACPI control method ***S***ource ***L***anguage. The programming language equivalent for ***AML***. ASL is compiled into AML images.

C0 Processor Power State:

While the processor is in this state, it executes instructions.

C1 Processor Power State

This processor power state has the lowest latency, The hardware latency on this state is required to be low enough that the operating software does not consider the latency aspect of the state when deciding whether to use it. Aside from putting the processor in a non-executing power state, this state has no other software-visible effects.

C2 Processor Power State:

The C2 state offers improved power savings over the C1 state. The worst-case hardware latency for this state is declared in the FACP Table and the operating software can use this information to determine when the C1 state should be used instead of the C2 state. Aside from putting the processor in a non-executing power state, this state has no other software-visible effects.

C3 Processor Power State:

The C3 state offers improved power savings of the C1 and C2 states. The worst-case hardware latency for this state is declared in the FACP Table, and the operating software can use this information to determine when the C2 state should be used instead of the C3 state. While in the C3 state, the processor's caches maintain state but ignore any snoops. The operating software is responsible for ensuring that the caches maintain coherency.

Control Method:

A control method is a definition of how the OS can perform a simple hardware task. For example, the OS invokes control methods to read the temperature of a thermal zone. Control methods are written in an encoded language called AML that can be interpreted and executed by the ACPI-compatible OS. An ACPI-compatible system must provide a minimal set of control methods in the ACPI tables. The OS provides a set of well-defined control methods that ACPI table developers can reference in their control methods. OEMs can support different revisions of chip sets with one BIOS by either including control

methods in the BIOS that test configurations and respond as needed or by including a different set of control methods for each chip set revision.

CPU, or processor:

The central processor unit (CPU), or processor, is the part of a platform that executes the instructions that do the work. An ACPI-compatible OS can balance processor performance against power consumption and thermal states by manipulating the processor clock speed and cooling controls. The ACPI specification defines a working state, labeled G0, in which the processor executes instructions. Processor low power states, labeled C1 through C3, are also defined. In the low power states the processor executes no instructions, thus reducing power consumption and, potentially, operating temperatures.

D0 - Fully-On:

This device power state is assumed to be the highest level of power consumption. The device is completely active and responsive, and is expected to remember all relevant context continuously.

D1:

The meaning of the D1 device power state is defined by each *class* of device; it may not be defined by many classes of devices. In general, D1 is expected to save less power and preserve more device context than D2.

D2:

The meaning of the D2 device power state is defined by each *class* of device; it may not be defined by many classes of devices. In general, D2 is expected to save more power and preserve less device context than D1 or D0. Buses in D2 may cause the device to loose some context (i.e., by reducing power on the bus, thus forcing the device to turn off some of its functions).

D3 - Off:

Power has been fully removed from the device. The device context is lost when this state is entered, so the OS software will reinitialize the device when powering it back on. Since device context and power are lost, devices in this state do not decode their addresses lines. Devices in this state have the longest restore times. All classes of devices define this state.

Definition Block:

A definition block contains information about hardware implementation and configuration details in the form of data and control methods, encoded in AML. An OEM can provide one or more definition blocks in the ACPI Tables. One definition block must be provided: the Differentiated Definition Block, which describes the base system. Upon loading the Differentiated Definition Block, the OS inserts the contents of the Differentiated Definition Block into the ACPI Name Space. Other definition blocks, which the OS can dynamically insert and remove from the active ACPI Name Space, can contain references to the Differentiated Definition Block.

Device:

Hardware components outside the core chip set of a platform. Examples of devices are LCD panels, video adapters, IDE CD-ROM and hard disk controllers, COM ports, etc. In the ACPI scheme of power management, buses are devices.

Differentiated System Description Table:

An OEM must supply a Differentiated System Description Table (DSDT) to an ACPI-compatible OS. The DSDT contains the Differentiated Definition Block, which supplies the implementation and configuration information about the base system. The OS always inserts the DSDT information into the ACPI Name Space at system boot time, and never removes it.

Embedded Controller and Embedded Controller Interface:

Embedded controllers are the general class of microcontrollers used to support OEM-specific implementations, mainly in mobile environments. The ACPI specification supports embedded controllers in any platform design, as long as the microcontroller conforms to one of the models described in this section. The embedded controller performs complex low-level functions, through a simple interface to the host microprocessor(s). ACPI defines a standard hardware and software communications interface between an OS driver and an embedded controller; this is the embedded controller interface. This interface allows any OS to provide a standard driver that can directly communicate with an embedded controller in the system, thus allowing other drivers within the system to communicate with and use the resources of system embedded controllers (for example, Smart Battery and AML code). This in turn enables the OEM to provide platform features that the OS and applications can use.

Firmware ACPI Control Structure:

The Firmware ACPI Control Structure (FACS) is a structure in read/write memory that the BIOS uses for handshaking between the firmware and the OS, and is passed to an ACPI-compatible OS via the Fixed ACPI Description Table (FACP). The FACS contains the system's hardware signature at last boot, the firmware waking vector, and the global lock.

Fixed ACPI Description Table:

An OEM must provide a Fixed ACPI Description Table (FACP) to an ACPI-compatible OS in the Root System Description Table. The FACP contains the ACPI Hardware Register Block implementation and configuration details the OS needs to direct management of the ACPI Hardware Register Blocks, as well as the physical address of the Differentiated System Description Table (DSDT) that contains other platform implementation and configuration details. The OS always inserts the name space information defined in the Differentiated Definition Block in the DSDT into the ACPI Name Space at system boot time, and the OS never removes it.

Fixed Platform Features, Fixed Feature Registers, and Fixed Feature Events

Fixed platform features are a set of features offered by an ACPI interface. The ACPI specification places restrictions on where and how the hardware programming model is generated. All fixed features, if used, are implemented as described in this specification so that the ACPI driver can directly access the fixed feature registers. The fixed feature registers are a set of hardware registers in fixed feature register space at specific address locations in system IO address space. ACPI defines *register blocks* for fixed features (each register block gets a separate pointer from the FACP ACPI table). Fixed feature events are a set of events that occur at the ACPI interface when a paired set of status and event bits in the fixed feature registers are set at the same time. While a fixed feature event occurs an SCI is raised. For ACPI fixed-feature events, the ACPI driver (or an ACPI-aware driver) acts as the event handler.

G0 - Working:

A computer state where the system dispatches user mode (application) threads and they execute. In this state, devices (peripherals) are dynamically having their power state changed. The user will be able to select (through some user interface) various performance/power characteristics of the system to have the software optimize for performance or battery life. The system responds to external events in real time. It is not safe to disassemble the machine in this state.

G1 - Sleeping:

A computer state where the computer consumes a small amount of power, user mode threads are *not* being executed, and the system "appears" to be off (from an end user's perspective, the display is off, etc.). Latency for returning to the Working state varies on the wakeup environment selected prior to entry of this state (for example, should the system answer phone calls, etc.). Work can be resumed without rebooting the OS because large elements of system context are saved by the hardware and the rest by system software. It is not safe to disassemble the machine in this state.

G2/S5 - Soft Off:

A computer state where the computer consumes a minimal amount of power. No user mode or system mode code is run. This state requires a large latency in order to return to the Working state. The system's context will not be preserved by the hardware. The system must be restarted to return to the Working state. It is not safe to disassemble the machine.

G3 - Mechanical Off:

A computer state that is entered and left by a mechanical means (e.g. turning off the system's power through the movement of a large red switch). This operating mode is required by various government agencies and countries. It is implied by the entry of this off state through a mechanical means that the no electrical current is running through the circuitry and it can be worked on without damaging the hardware or endangering the service personnel. The OS must be restarted to return to the Working state. No hardware context is retained. Except for the real time clock, power consumption is zero.

Generic Platform Features and General Purpose Event (GPE) Registers:

A generic feature of a platform is value-added hardware implemented through control methods and general-purpose events. The general purpose event (GPE) registers contain the event programming model for generic features. All generic events generate SCIs.

Global System States:

Global system states apply to the entire system, and are visible to the user. The various global system states are labeled G0 through G3 in the ACPI specification.

Legacy State, Legacy Hardware, and Legacy OS:

Legacy state is a computer state where power management policy decisions are made by the platform hardware/firmware shipped with the system. The legacy power management features found in today's systems are used to support power management in a system that uses a legacy OS that does not support the OS-directed power management architecture. Legacy hardware is a computer system that has no ACPI or OSPM power management support. A legacy OS is an operating system that is not aware of and does not direct power management functions of the system. Included in this category are operating systems with APM 1.x support.

Multiple APIC Description Table:

The Multiple APIC Description Table (APIC) is used on systems supporting the APIC to describes the APIC implementation. Following the Multiple APIC Description Table is a list of APIC structures that declare the APIC features of the machine.

Object:

The nodes of the ACPI Name Space are objects inserted in the tree by the OS using the information in the system definition tables. These objects can be data objects, package objects, control method objects, etc. Package objects refer to other objects. Objects also have type, size, and relative name.

Object name:

Object names are part of the ACPI Name Space. There is a set of rules for naming objects.

***Package*:**

A set of objects.

Persistent System Description Table:

Persistent System Description Tables are Definition Blocks, similar to Secondary System Description Tables, except a Persistent System Description Table can be saved by the OS and automatically loaded at every boot.

Power Button:

A user push button that switches the system from the sleeping/soft off state to the working state, and signals the OS to transition to a sleeping/soft off state from the working state.

Power Resources:

Power resources are resources (for example, power planes and clock sources) that a device requires to operate in a given power state.

Power Sources:

The battery and AC adapter that supply power to a platform.

Root System Description Pointer:

An ACPI compatible system must provide a Root System Description Pointer in the systems low address space. This structure's only purpose is to provide the physical address of the Root System Description Table.

Root System Description Table:

The Root System Description Table starts with the signature 'RSDT,' followed by an array of physical pointers to the other System Description Tables that provide various information on other standards that are defined on the current system. The OS locates that Root System Description Table by following the pointer in the Root System Description Pointer structure.

S1 Sleeping State:

The S1 sleeping state is a low wake-up latency sleeping state. In this state, no system context is lost (CPU or chip set) and hardware maintains all system context.

S2 Sleeping State

The S2 sleeping state is a low wake-up latency sleeping state. This state is similar to the S1 sleeping state except the CPU and system cache context is lost (the OS is responsible for maintaining the caches and CPU context). Control starts from the processor's reset vector after the wake-up event.

S3 Sleeping State:

The S3 sleeping state is a low wake-up latency sleeping state where all system context is lost except system memory. CPU, cache, and chip set context are lost in this state. Hardware maintains memory context and restores some CPU and L2 configuration context. Control starts from the processor's reset vector after the wake-up event.

S4 Sleeping State:

The S4 sleeping state is the lowest power, longest wake-up latency sleeping state supported by ACPI. In order to reduce power to a minimum, it is assumed that the hardware platform has powered off all devices. Platform context is maintained.

S4 - Non-Volatile Sleep:

S4 Non-Volatile Sleep (NVS) is a special global system state that allows system context to be saved and restored (relatively slowly) when power is lost to the motherboard. If the system has been commanded to enter S4, the OS will write all system context to a non-volatile storage file and leave appropriate context markers. The machine will then enter the S4 state. When the system leaves the Soft Off or Mechanical Off

state, transitioning to Working (G0) and restarting the OS, a restore from a NVS file can occur. This will only happen if a valid NVS data set is found, certain aspects of the configuration of the machine has not changed, and the user has not manually aborted the restore. If all these conditions are met, as part of the OS restarting it will reload the system context and activate it. The net effect for the user is what looks like a resume from a Sleeping (G1) state (albeit slower). The aspects of the machine configuration that must not change include, but are not limited to, disk layout and memory size. It might be possible for the user to swap a PC Card or a Device Bay device, however.

Note that for the machine to transition directly from the Soft Off or Sleeping states to S4, the system context must be written to non-volatile storage by the hardware; entering the Working state first so the OS or BIOS can save the system context takes too long from the user's point of view. The transition from Mechanical Off to S4 is likely to be done when the user is not there to see it.

Because the S4 state relies only on non-volatile storage, a machine can save its system context for an arbitrary period of time (on the order of many years).

S5 Soft Off State:

The S5 state is similar to the S4 state except the OS does not save any context nor enable any devices to wake the system. The system is in the "soft" off state and requires a complete boot when awakened. Software uses a different state value to distinguish between the S5 state and the S4 state to allow for initial boot operations within the BIOS to distinguish whether or not the boot is going to wake from a saved memory image.

Secondary System Description Table:

Secondary System Description Tables are a continuation of the Differentiated System Description Table. Multiple Secondary System Description Tables can be used as part of a platform description. After the Differentiated System Description Table is loaded into ACPI name space, each secondary description table with a unique OEM Table ID is loaded. This allows the OEM to provide the base support in one table, while adding smaller system options in other tables. Note: Additional tables can only add data, they cannot overwrite data from previous tables.

Sleep Button:

A user push button that switches the system from a sleeping state to the working state, and signals the OS to transition to a sleeping state from the working state.

Smart Battery Subsystem and Smart Battery Table:

A battery subsystem that conforms to the following specifications: --battery, charger, selector list—and the additional ACPI requirements. The Smart Battery table is an ACPI table used on platforms that have a Smart Battery Subsystem. This table indicates the energy levels trip points that the platform requires for placing the system into different sleeping states and suggested energy levels for warning the user to transition the platform into a sleeping state.

SMBus and SMBus Interface:

SMBus is a two-wire interface based upon the I²C protocol. The SMBus is a low-speed bus that provides positive addressing for devices, as well as bus arbitration. ACPI defines a standard hardware and software communications interface between an OS bus driver and an SMBus Controller via an embedded controller; this is the SMBus interface.

System Control Interrupt (SCI):

A system interrupt used by hardware to notify the OS of ACPI events. The SCI is a active low, shareable, level interrupt.

System Management Interrupt (SMI):

An OS-transparent interrupt generated by interrupt events on legacy systems. By contrast, on ACPI systems, interrupt events generate an OS-visible interrupt that is shareable (edge-style interrupts will not work). Hardware platforms that want to support both legacy operating systems and ACPI systems must support a way of re-mapping the interrupt events between SMIs and SCIs when switching between ACPI and legacy models.

Thermal States:

Thermal states represent different operating environment temperatures within thermal zones of a system. A system can have one or more thermal zones; each thermal zone is the volume of space around a particular temperature sensing device. The transitions from one thermal state to another are marked by trip points, which are implemented to generate a System Control Interrupt (SCI) when the temperature in a thermal zone moves above or below the trip point temperature.